Building Java Programs

Stuart Reges and Marty Stepp

Table of Contents

Chapter 1: Introduction to Java Programming

- 1.1 Basic Computing Concepts
  - Why Programming?
  - Hardware and Software
  - The Digital Realm
  - Why Java?
  - The Process of Programming
  - The Java Programming Environment
- 1.2 And Now--Java
  - String Literals (Strings)
  - System.out.println
  - Escape Sequences
  - Identifiers and Keywords
  - Comments and Readability
  - A Complex Example: DrawFigures1
- 1.3 Program Errors
  - Syntax Errors
  - Logic Errors (bugs)
- 1.4 Procedural Decomposition
  - Static Methods
  - Flow of Control
  - Methods That Call Other Methods
  - An Example Runtime Error
- 1.5 Case Study: DrawFigures
  - Structured Version
  - Final Version without Redundancy
  - Analysis of Flow of Execution

Chapter 2: Primitive Data and Definite Loops

- 2.1 Basic Data Concepts
  - Primitive Types
  - Expressions
  - Literals
  - Arithmetic Operators
  - Precedence
  - Mixing Types and Casting
- 2.2 Variables
  - Assignment/Declaration Variations
  - String Concatenation
  - Increment/Decrement Operators
  - Variables and Mixing Types
- 2.3 The for Loop
  - Tracing for Loops

    - print Versus println
    - Nested for Loops
- 2.4 Managing Complexity
    - Scope
    - Pseudocode
    - A Decrementing for Loop
    - Class Constants
- 2.5 Case Study: A Complex Figure
    - Problem Decomposition and Pseudocode
    - Line Pattern Table
    - Initial Structured Version
    - Adding a Class Constant
    - Further Variations

Chapter 3: Introduction to Parameters and Objects

- 3.1 Parameters
    - The Mechanics of Parameters
    - Limitations of Parameters
    - Multiple Parameters
    - Parameters Versus Constants
    - Overloading of Methods
- 3.2 Methods that Return Values
    - The Math Class
    - Defining Methods that Return Values
- 3.3 Using Objects
    - String Objects
    - Point Objects
    - Reference Semantics
    - Multiple Objects
    - Objects as Parameters to Methods
- 3.4 Interactive Programs
    - Scanner Objects
    - A Sample Interactive Program
- 3.5 Case Study: Projectile Trajectory
    - An Unstructured Solution
    - A Structured Solution

Supplement 3G: Graphics (optional)

- 3G.1 Introduction to Graphics
    - DrawingPanel
    - Lines and Shapes
    - Colors
    - Text and Fonts
- 3G.2 Procedural Decomposition with Graphics
    - A Larger Example: DrawDiamonds
    - Summary of Graphics Methods

- 3G.3 Case Study: Pyramids
  - An Unstructured Solution
  - Generalizing the Drawing of Pyramids
  - A Complete Structured Solution

Chapter 4: Conditional Execution

- 4.1 Loop Techniques
  - Cumulative Sum
  - Fencepost Loops (aka "loop and a half")
- 4.2 if/else Statements
  - Relational Operators
  - Cumulative Sum with if
  - Fencepost with if
  - Nested if/else
- 4.3 Subtleties of Conditional Execution
  - Object Equality
  - Roundoff Errors
  - Factoring if/else Statements
  - Min/Max Loops
- 4.4 Text Processing
  - The char Type
  - System.out.printf
- 4.5 Methods with Conditional Execution
  - Preconditions and Postconditions
  - Throwing Exceptions
  - Revisiting Return Values
- 4.6 Case Study: Body Mass Index (BMI)
  - One-person Unstructured Solution
  - Two-person Unstructured Solution
  - Two-person Structured Solution

Chapter 5: Program Logic and Indefinite Loops

- 5.1 The while Loop
  - A Loop to Find the Smallest Divisor
  - Sentinel Loops
  - Random Numbers
- 5.2 The boolean Type
  - Logical Operators
  - Short-Circuited Evaluation
  - boolean Variables and Flags
  - Boolean Zen
- 5.3 User Errors
  - Scanner Lookahead
  - Handling User Errors
- 5.4 Indefinite Loop Variations
  - The do/while Loop

    - Break and "forever" Loops
- 5.5 Assertions and Program Logic
    - Reasoning About Assertions
    - A Detailed Assertions Example
    - The Java assert Statement
- 5.6 Case Study: NumberGuess
    - Initial Version without Hinting
    - Randomized Version with Hinting
    - Final Robust Version

Chapter 6: File Processing

- 6.1 File Reading Basics
    - Data, Data Everywhere
    - File Basics
    - Reading a File with a Scanner
- 6.2 Details of Token-Based Processing
    - Structure of Files and Consuming Input
    - Scanner Parameters
    - Paths and Directories
    - A More Complex Input File
- 6.3 Line-Based Processing
    - String Scanners and Line/Token Combinations
- 6.4 Advanced File Processing
    - Output Files with PrintStream
    - Try/Catch Statements
- 6.5 Case Study: Weighted GPA

Chapter 7: Arrays

- 7.1 Array Basics
    - Constructing and Accessing an Array
    - A Useful Application of Arrays
    - Random Access
    - Arrays and Methods
    - The For-Each Loop
    - Limitations of Arrays
- 7.2 Advanced Arrays
    - Shifting Values in an Array
    - Initializing Arrays
    - Arrays in the Java Class Libraries
    - Arrays of Objects
    - Command Line Arguments
- 7.3 Multidimensional Arrays (optional)
    - Rectangular Two-Dimensional Arrays
    - Jagged Arrays
- 7.4 Case Study: Hours Worked
    - The transferFrom Method

        - The sum Method
        - The addTo Method
        - The print Method
        - The Complete Program

Chapter 8: Defining Classes

- 8.1 Object-Oriented Programming Concepts
    - Classes and Objects
- 8.2 Object State: Fields
- 8.3 Object Behavior: Methods
    - A Detailed Example
    - Mutators and Accessors
- 8.4 Object Initialization: Constructors
- 8.5 Encapsulation
    - Private Data Fields
    - Class Invariants
- 8.6 More Instance Methods
    - The toString Method
    - The equals Method
- 8.7 The this Keyword
    - Multiple Constructors
- 8.8 Case Study: Designing a Stock Class
    - Stock Behavior
    - Stock Fields
    - Stock Constructor
    - Stock Method Implementation
    - The Complete Stock Class

Chapter 9: Inheritance and Interfaces

- 9.1 Inheritance Concepts
    - Non-programming Hierarchies
- 9.2 Programming with Inheritance
    - Overriding Methods
    - Polymorphism
- 9.3 The Mechanics of Polymorphism
    - Diagramming Polymorphic Code
- 9.4 Interacting with the Superclass
    - Inherited Fields
    - Calling a Superclass's Constructor
    - Calling Overridden Methods
    - A Larger Example: Point3D
- 9.5 Inheritance in the Java Class Libraries
    - Graphics2D (optional)
    - Input/Output Streams
- 9.6 Interfaces
    - An Interface for Shape Classes

    - Implementing the Shape Interface
    - Benefits of Interfaces
    - Interfaces in the Java Class Libraries
- 9.7 Case Study: Designing a Hierarchy of Financial Classes
    - Class Design
    - Initial Redundant Implementation
    - Abstract Classes

Chapter 10: ArrayLists

- 10.1 ArrayLists
    - Basic ArrayList Operations
    - ArrayList Searching Methods
    - Sample ArrayList Problems
    - The for-each Loop
    - Wrapper Classes
- 10.2 The Comparable Interface
    - Natural Ordering and compareTo
    - Implementing Comparable
- 10.3 Case Study: Vocabulary Comparison
    - Version 1: Compute Vocabulary
    - Version 2: Compute Overlap
    - Version 3: Complete Program

Chapter 11: Java Collections Framework

- 11.1 Lists
    - Collections
    - LinkedList versus ArrayList
    - Iterators
    - LinkedList Example: Sieve
    - Abstract Data Types (ADTs)
- 11.2 Sets
    - Set Concepts
    - TreeSet versus HashSet
    - Set Operations
    - Set Example: Lottery
- 11.3 Maps
    - Basic Map Operations
    - Map Views (keySet and values)
    - TreeMap versus HashMap
    - Map Example: WordCount
    - Collection Overview

Chapter 12: Recursion

- 12.1 Thinking Recursively

    - A Nonprogramming Example
    - An Iterative Solution Converted to Recursion
    - Structure of Recursive Solutions
- 12.2 A Better Example of Recursion
    - Mechanics of Recursion
- 12.3 Recursive Functions
    - Integer Exponentiation
    - Greatest Common Divisor
- 12.4 Recursive Graphics (optional)
- 12.5 Case Study: Prefix Evaluator
    - Infix, Prefix and Postfix Notation
    - Prefix Evaluator
    - Complete Program

Chapter 13: Searching and Sorting

- 13.1 Searching and Sorting in the Java Class Libraries
    - Binary Search
    - Sorting
    - Shuffling
    - Custom Ordering with Comparators
- 13.2 Program Efficiency
    - Algorithm Runtimes
    - Complexity Classes
- 13.3 Implementing Searching Algorithms
    - Sequential Search
    - Binary Search
    - Searching Objects
- 13.4 Implementing Sorting Algorithms
    - Selection Sort
    - Merge Sort
    - Other Sorting Algorithms

Chapter 14: Graphical User Interfaces

- 14.1 Graphical Input and Output with JOptionPane
- 14.2 Graphical Components
    - Working with JFrames
    - Common Components: Buttons, Labels, and Text Fields
    - JTextArea, JScrollPane, and Font
    - Icons
- 14.3 Laying Out Components in a Frame
    - Layout Managers
    - SpringLayout
    - Composite Layout
- 14.4 Events
    - Action Events and ActionListener
    - More Sophisticated ActionEvents

        - A Larger GUI Example with Events: Credit Card GUI
        - Mouse Events
    - 14.5 2D Graphics
        - Drawing Onto Panels
        - Simple Animation with Timers
    - 14.6 Case Study: Demystifying DrawingPanel

Appendix A: Answers to Self-Check Problems

Chapter 1
Introduction to Java Programming

Copyright © 2006 by Stuart Reges and Marty Stepp

- 1.1 Basic Computing Concepts
  - Hardware and Software
  - The Digital Realm
  - Why Programming?
  - Why Java?
  - The Process of Programming
  - The Java Programming Environment
- 1.2 And Now--Java
  - String Literals (Strings)
  - System.out.println
  - Escape Sequences
  - Identifiers and Keywords
  - Comments and Readability
  - A Complex Example: DrawFigures1
- 1.3 Program Errors
  - Syntax Errors
  - Logic Errors (bugs)
- 1.4 Procedural Decomposition
  - Static Methods
  - Flow of Control
  - Methods That Call Other Methods
  - An Example Runtime Error
- 1.5 Case Study: DrawFigures
  - Structured Version
  - Final Version without Redundancy
  - Analysis of Flow of Execution

Introduction

In this chapter, we introduce some basic terminology about computers and programming. We discuss the Java language and its programming environment. We take a brief look at writing simple but structured Java programs that produce output.

1.1 Basic Computing Concepts

Computers are pervasive in our daily lives, giving us access to nearly limitless information. Some of this information is essential news, like the headlines at cnn.com. Some of it is more frivolous: If you're concerned about whether the guy you met last night cheats on his girlfriends, perhaps you've visited dontdatehimgirl.com. Computers let us share photos with our families and map directions to the nearest pizza place for dinner.

Lots of real-world problems are being solved by computers, some of which don't much resemble the one on your desk or lap. The human genome is sequenced and searched for DNA patterns using powerful computers. There are computers in recently manufactured cars, monitoring each vehicle's status and motion. Digital music players such as Apple's iPod are actually computers underneath their small casing. Even the Roomba vacuum cleaning robot houses a computer with complex instructions about how to dodge furniture while cleaning your floors.

But what makes a computer a computer? Is a calculator a computer? Is a human being with a paper and pencil a computer? The next several sections attempt to address this question while leading us toward putting computers in our command through programming.

Hardware and Software

A *computer* is a machine that manipulates data and executes lists of instructions known as *programs*.

Program

A list of instructions to be carried out by a computer.

One key feature that differentiates a computer from a simpler machine like a calculator is its versatility. The same computer can perform many different tasks (playing games, computing income taxes, connecting to other computers around the world) depending on what program it is running at a given moment. A computer can run not only the programs that exist on it currently but also new programs that haven't even been written yet.

The physical components that make up a computer are collectively called *hardware*. One of the most important pieces of hardware is the central processing unit or *CPU*, which is the brain of the computer that executes instructions. Also important is the computer's *memory* (often called random access memory or RAM, because the computer can access any part of that memory at any time). The computer uses its memory to store programs that are being executed, along with their data. RAM is limited in size and does not retain its contents when the computer is turned off. Therefore, computers generally also use a *hard disk* as a larger permanent storage area.

Computer programs are collectively called *software*. The primary piece of software running on a computer is its operating system. An *operating system* provides an environment where many application programs may be run at the same time, as well as providing a bridge between those programs and the hardware and user. The programs that run inside the operating system are often called *applications*.

When the user selects a program to be run by the operating system (such as by double-clicking the icon of that program on the desktop), several things happen. The instructions for that program are loaded into the computer's memory from the hard disk. The operating system allocates memory for that program to use. The instructions of the program are fed from the memory to the CPU and executed sequentially.

The Digital Realm

In the last section we saw that a computer is a general purpose device that can be programmed. You will often hear people refer to modern computers as *digital* computers because of the way that they operate.

Digital

Based on numbers that increase in discrete increments such as the integers (0, 1, 2, 3, etc).

Because computers are digital, everything that is stored on a computer is stored as a sequence of integers. This includes every program and every piece of data. The idea of representing everything as an integer was fairly unusual in the 1940's when the first computers were built. This idea seems less unusual today now that we have digital music, digital pictures and digital movies. An mp3 file, for example, is simply a long sequence of integers that stores audio information.

Not only are computers digital, storing all information as integers, they are also *binary*, which means that those integers are stored as *binary numbers*.

Binary Number

A number composed of just 0's and 1's, also known as a base-2 number.

Humans generally work with *decimal* or base-10 numbers. It might seem odd that computers use binary numbers when people are so used to base-10 numbers, but we use base-10 because it matches our physiology (10 fingers and 10 toes). In the case of computers, we want a system that will be easy to create and that will be very reliable. It turns out to be simpler to build a system on top of binary phenomena (e.g., a circuit being open or closed) than to build it on a system with ten different states to distinguish (e.g., 10 different voltage levels).

From a mathematical point of view, you can store things just as easily using binary numbers as you can using base-10 numbers. Since it is easier to construct the physical device using binary numbers, that's what computers use.

It does mean, however, that people who aren't used to computers often encounter unfamiliar conventions. As a result, it is worth spending a little time reviewing how binary numbers work. In binary you start with 0 and you can count up, just like you do in base-10, but you run out of digits much faster. So counting in binary you say:

```
0
1
```

And already you've run out of digits. This is like reaching 9 when you count in base 10. After you run out of digits, you carry over to the next digit. So the next two numbers are:

```
10
11
```

And again we have run out of digits. This is like reaching 99 in base 10. So again we carry into the next digit to form the 3-digit number 100. In binary, whenever you see a series of ones as in 111111, you know you're just one away from the digits all flipping to 0 with a 1 in front, in the same way that in base-10 when you see a number like 999999, you know that you are one away from those digits turning to 0 with a 1 in front. The following table shows how we would count up to the base-10 number 16 using binary.

decimal	binary	decimal	binary
0	0		
1	1	9	1001
2	10	10	1010
3	11	11	1011
4	100	12	1100
5	101	13	1101
6	110	14	1110
7	111	15	1111
8	1000	16	10000

There are several useful observations to make about binary numbers. Notice in the table above that the binary numbers 1, 10, 100, 1000, 10000 are all perfect powers of 2 (2^0, 2^1, 2^2, 2^3, 2^4). In the same way that in base-10 we talk about a one's digit, ten's digit, hundred's digit and so on, we can think in binary of a one's digit, two's digit, four's digit, eight's digit, sixteen's digit and so on.

Computer scientists quickly found themselves needing to refer to the sizes of different binary quantities so we invented the term *bit* to refer to a single binary digit and the term *byte* to refer to 8 bits. To talk about large amounts of memory, we talk about kilobytes (KB), megabytes (MB), gigabytes (GB) and so on. Many people think that these correspond to the metric system where "kilo" means 1000, but that is only approximately true. We use the fact that 2^{10} is appromixately equal to 1000 (it actually equals 1024). So a kilobyte is 2^{10} bytes (1024 bytes), a megabyte is 2^{20} bytes (1,048,576 bytes), a gigabyte is 2^{30} bytes (1,073,741,824 bytes) and so on.

Why Programming?

At most universities the first course in computer science is predominantly a programming course. Many computer scientists are bothered by this because it leaves people with the impression that computer science = programming. While it is true that many trained computer scientists spend time programming, there is a lot more to the discipline than just programming. So why do we study programming first?

A Stanford computer scientist named Don Knuth answers this question by saying that the common thread to most of computer science is that we all in some way work with *algorithms*.

Algorithm

A step-by-step description of how to accomplish a task.

Knuth is an expert in algorithms, so he would naturally be biased to think of them as the center of computer science. He claims that what is most important is not the algorithms themselves, but rather the thought process that computer scientists employ. Knuth has said:

> It has often been said that a person does not really understand something until after teaching it to someone else. Actually a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm.

Knuth is describing a thought process that is common to most of computer science which he refers to as *algorithmic thinking*. So we study programming not because it is the most important aspect of computer science, but because it is the best way to explain the approach computer scientists take to solving problems.

Algorithms are expressed as computer programs.

The Process of Programming

The word *code* describes program fragments ("these four lines of code") or the act of programming ("Let's code this into Java"). Once a program has been written, you can *execute* it.

Program Execution

The act of carrying out the instructions contained in a program.

The process of execution is often called *running*. It can be used as a verb, "When my program runs it does something strange. . .," or as a noun, "The last run of my program produced these results. . ."

Computer programs are stored internally as a series of binary numbers known as the *machine language* of the computer. In the early days programmers entered numbers like these directly into the computer. Obviously this is a tedious and confusing way to program a computer and we have invented all sorts of mechanisms to simplify this process.

Modern programmers write in what are known as high-level programming languages like Java. Such programs cannot be run directly on a computer. They first have to be translated into a different form by a special program known as a *compiler*.

Compiler

A program that translates a computer program written in one language into an equivalent program in another language (often, but not always, translating into machine language).

A compiler that translates directly into machine language creates a program that can be executed directly on the computer. We refer to such a program as an executable and we refer to such compilers as *native compilers* because they compile code to the lowest possible level (the native machine language of the computer).

This approach works well when you know exactly what computer you want your program to execute on. But what if you want to execute a program on many different computers? Using this approach you'd need a compiler that generates different machine language output for each different computer. The designers of Java decided to use a different approach. They cared a lot about being able to run on many different computers because they wanted to have a language that worked well for the web. People who write applets (Java programs that live inside web pages) want those programs to run on many different computers.

Instead of compiling into machine language, Java programs are compiled into what are known as *Java bytecodes*. These bytecodes represent an intermediate level. They aren't quite as high-level as Java but they also aren't quite as low-level as machine language. The key thing is that one set of bytecodes can execute on many different machines. Java bytecodes are similar to machine language. In fact, they are the machine language of a theoretical computer known as the Java Virtual Machine or JVM.

Java Virtual Machine (JVM)

A theoretical computer whose machine language is the set of Java bytecodes.

This isn't an actual machine but it's similar to actual machines. By compiling down to this level, there isn't as much work left to turn the Java bytecodes into actual machine instructions.

In the Java programming language, nothing can exist outside of what is called a class.

Class

A unit of code that is the basic building block of Java programs.

The notion of a class is much richer than this as we'll see when we get to Chapter 8, but for now all we need to know is that each of our Java programs will be stored in a class.

To actually execute a Java class file, you need another program that will execute the Java bytecodes. Such programs are known generically as *Java runtimes* and the standard environment distributed by Sun is known as the Java Runtime Environment.

Java Runtime Environment (JRE)

A program that executes compiled Java class files.

Most people have a Java runtime on their computer even if they don't know about it. For example, Apple's OS X includes a Java runtime and the standard Windows installer from Microsoft installs a Java runtime.

Why Java?

When Sun Microsystems released Java in 1995, they published a document called a "white paper" describing their new programming language. Perhaps the key sentence from that paper is the following:

> Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.

This sentence covers many of the reasons we'll be using Java as our programming language in this textbook. Java is reasonably simple for beginners to learn. Java embraces object-oriented programming, a style of writing programs that has been shown to be very successful for creating large and complex software.

Java also includes a large amount of pre-written software that programmers can utilize to enhance their programs. Such off-the-shelf software components are often called *libraries*. For example, if you wish to write a program that connects to a site on the internet, Java contains a library to simplify the connection for you. Java contains libraries to draw graphical user interfaces, retrieve data from databases, and perform complex mathematical computations, among many other things. These libraries collectively are called the *Java class libraries*.

Java Class Libraries

Java's collection of pre-existing code that provides solutions to common programming problems.

The richness of Java's class libraries are an extremely important factor in the rise of Java as a popular language. As of version 1.5, the class libraries include over 3200 entries.

Another reason to use Java is that it has a vibrant programmer community. There is a large amount of online documentation and tutorials available to help programmers learn new skills. Many of these documents are written by Sun themselves, such as an extensive reference to Java's class libraries called the *API Specification* (where "API" stands for Application Programming Interface).

Java is extremely platform-independent; unlike programs written in many other languages, the same Java program can be executed on many different operating systems such as Windows, Linux, and Macintosh.

Java is used extensively by programmers for both research and business applications. This means that a large number of programming jobs exist in the marketplace today for skilled Java programmers. A sample Google search for the phrase "Java jobs" returns 124,000,000 hits.

The Java Programming Environment

You must become familiar with your computer setup before you start programming. Each computer provides a different environment for program development, but there are some common elements that deserve comment. No matter what environment you use, you will follow the same basic three steps:

1. type in a program as a Java class
2. compile the program file
3. run the compiled version of the program

The basic unit of storage on most computers is a file. Every file has a name. A file name ends with an *extension*, which is the part of a file's name that follows the period. A file's extension indicates the type of data contained in the file. For example, files with extension `.doc` are Microsoft Word documents, and files with extension `.mp3` are MP3 audio files.

You will create files whose contents are Java programs. Java program files must use the extension `.java`. When you compile a Java program, the resulting Java bytecodes are stored in a file with the same name and the extension `.class`.

Most Java programmers use what are known as Integrated Development Environments or IDEs that provide an all-in-one environment for creating, editing, compiling and executing program files. Some of the more popular choices for introductory computer science classes are Eclipse, DrJava, BlueJ and TextPad. Your instructor will tell you what environment you should use.

For example, you might type in the following program file:

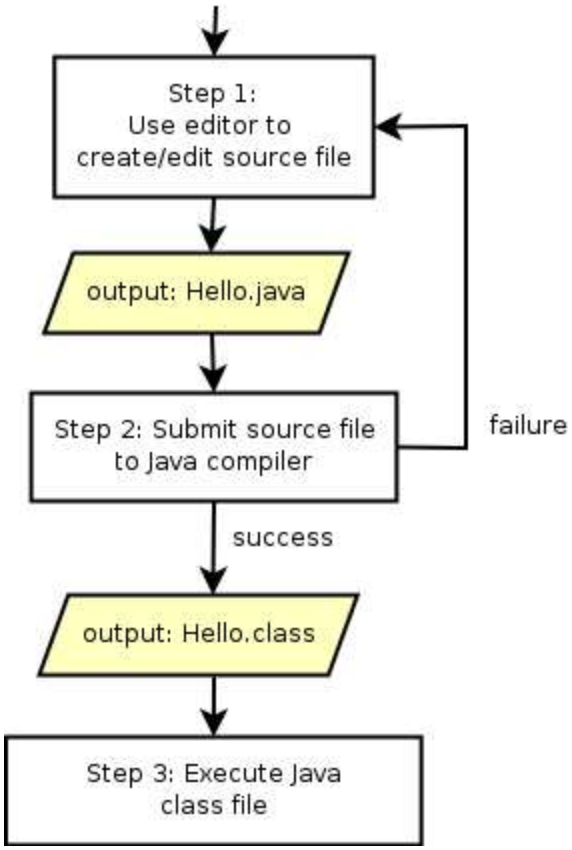
```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4      }
5  }
```

Don't worry about the details of this program right now. We will explore that in the next section.

Once you have typed in a program file, you move to step 2 and compile it. The command to compile will be different in each different development environment, but the process is the same. You have to submit your class file to the compiler for translation (typical commands are "compile" or "build"). There might be errors, in which case you'd have to go back to the editor and fix the errors and try to compile again. We'll discuss errors in more detail later in this chapter.

Once you have successfully compiled your program, you are ready to move to step 3 by running the program. Again, the command to do this will differ from one environment to the next, but the process is similar (the typical command is "run").

The following diagram summarizes the steps we would follow in creating a program named Hello.java:



The Hello.java program involves an onscreen window known as the *console*.

Console Window

A special text-only window in which Java programs interact with the user.

The console window is a classic way to interact with computers where the computer displays text on the screen and sometimes waits for the user to type responses. This is known as console or terminal interaction. The text typed by the computer in the console window is known as the *output* of the program. Anything typed by the user is known as the console *input*.

To keep things simple, most of the sample programs in this book involve console interaction. Keeping the interaction simple will allow us to focus our attention and effort on other aspects of programming. For those who are interested, Chapter 10 describes how to write programs that use a more modern kind of interface known as a Graphical User Interface or GUI.

1.2 And Now--Java

It's time to look at a complete Java program. It is a tradition in computer science that when you describe a new programming language, you should start with a program that produces a single line of output with the words, "Hello, world!" The hello world tradition has been broken by many authors of Java books because the program turns out not to be as short and simple when written in Java.

Here is our hello world program:

```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4      }
5  }
```

This defines a class called Hello. Sun has established the convention that class names always begin with a capital letter, which makes it easy to recognize that something is the name of a class. Java requires that the class name and the file name match, so this would have to be stored in a file called Hello.java. Fortunately, you don't have to understand all of the details of this program just yet. But you do need to understand the basic structure.

Remember that the basic unit of code in Java is the class. Every program must be contained within a Java class. The basic form of a Java class is as follows.

```
public class <name> {
    <method>
    <method>
    ...
    <method>
}
```

The description above is known as a *syntax template* because it describes the basic form of a Java construct. Java has rules that determine its legal *syntax* or grammar. Each time we see a new element of Java, we'll begin by looking at its syntax template. By convention, we use the characters less-than ("<") and greater-than (">") in a syntax template to indicate items that need to be filled in. In this case the name of the class and the methods both need to be filled in. When we write "..." in a list of elements, we're indicating that any number of those elements may be included.

The first line of the class is known as the *class header*. The word `public` in the header indicates that this class is available to anyone to use. Notice that the program code in a class is enclosed in curly brace characters. The curly brace characters are used in Java to group together related bits of code.

Grouping Characters

The curly brace characters `{` and `}` used in Java to group together related lines of code.

In this case, the curly braces are indicating that everything defined inside these braces is part of this public class.

So what exactly can appear inside the curly braces? What can be contained in a class? All sorts of things, but for now, we'll limit ourselves to what are known as *methods*. Methods are the next smallest unit of code in Java and they represent a single action or calculation to be performed.

Method

A program unit that represents a particular action or computation.

Simple methods are like verbs. They command the computer to perform some action. Inside the curly braces for a class, you can define several different methods. At a minimum, a complete program requires a special method that is known as the `main` method. It has the following syntax:

```
public static void main(String[] args) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

Just as the first line of a class is known as a class header, the first line of a method is known as a *method header*. The header for main is rather complicated. Most people memorize this as a kind of magical incantation. You want to open the door to Ali Baba's cave? You say, "Open sesame." You want to create an executable Java program? You say, "`public static void main(String[] args)`". A group of Java teachers makes fun of this with a web site called www.publicstaticvoidmain.com.

Memorizing magical incantations is never satisfying, especially for computer scientists who like to know everything that is going on in their programs. But this is a place where Java shows its ugly side, and we just have to live with it. New programmers, like new drivers, must learn to use something complex without fully understanding how it works. Fortunately, by the time you finish this book, you'll understand every part of the incantation.

Notice that the main method has a set of curly braces of its own. They are again used for grouping, saying that everything that appears between the braces is part of the main method. The lines in between the curly braces specify the series of actions to perform in executing the program. We refer to these as the *statements* of the program. Just as you put together an essay by stringing together complete sentences, you put together a method by stringing together statements.

Statement

An executable snippet of code that represents a complete command.

The sample "hello world" program has just a single statement that is known as a `println` statement:

```
System.out.println("Hello, world!");
```

Notice that this statement ends with a semicolon. The semicolon has a special status in Java. It is used to terminate statements in the same way that periods terminate sentences in English.

Statement Terminator

The semicolon character ; used in Java to terminate statements.

In the "hello world" program there is just a single command to produce a line of output, but consider the following variation called Hello3 that has three lines of code to be executed in the main method.

```
1  public class Hello3 {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4          System.out.println();
5          System.out.println("This program produces three lines of output.");
6      }
7  }
```

Notice that there are three semicolons in the main method, one at the end of each of the three println statements. The statements are executed in the order in which they appear from first to last, so the program above produces the following output.

```
Hello, world!

This program produces three lines of output.
```

Let's summarize the different levels we just looked at:

- A Java program is stored in a class.
- Inside the class you can include methods. At a minimum, a complete program requires that you have a special method called main.
- Inside a method like main you include a series of statements that each represent a single command for the computer to execute.

It may seem odd to put the opening curly brace at the end of a line rather than on a line by itself. Some people would use this style of indentation for the program instead:

```
1  public class Hello2
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Hello, world!");
6      }
7  }
```

Different people will make different choices about the placement of curly braces. The style we use follows Sun's official Java coding conventions. But the other style has its advocates. Often people will passionately argue that one way is much better than the other, but really these are matters of personal taste because each choice has some advantages and some disadvantages. Your instructor may require a particular style; if not, you should choose a style that you are comfortable with and then use it consistently.

Now that we have seen an overview of the structure, let's examine some of the details of Java programs in greater detail.

Did You Know: Hello World

The "hello world" tradition was started by Brian Kernighan and Dennis Ritchie who invented a programming language known as C in the 1970's. The first complete program in their 1978 book describing the C language was a "hello world" program. Kernighan and Ritchie and their book *The C Programming Language* have both been affectionately referred to as "K & R" ever since.

Many major programming languages have borrowed the basic C syntax as a way to leverage the popularity of C and to encourage programmers to switch. The languages C++ and Java both borrow a great deal of their core syntax from C.

Kernighan and Ritchie also had a distinctive style for where to place curly braces and how to indent their programs that has become known as "K & R style." This is the style that Sun recommends and that we use in this book.

String Literals (Strings)

Often in writing Java programs (such as the preceding "hello world" program), we want to include some literal text that we want to send to the console window as output. Programmers have traditionally referred to such text as a *string* because it is composed of a sequence of characters that we string together. The Java language specification refers to these as *string literals*, although programmers often simply refer to them as strings.

In Java you specify a string literal by surrounding the literal text in quotation marks, as in:

```
"This is a bunch of text surrounded by quotation marks."
```

You must use double quotation marks, not single quotation marks. The following is not a valid string literal.

```
'Bad stuff here.'
```

But the following is a valid string literal.

```
"This is a quote even with 'these' quotes inside."
```

String literals must not span more than one line of a program. The following is not a valid string literal.

```
"This is really
bad stuff
right here."
```

System.out.println

As you have seen, the main method of a Java program contains a series of statements for the computer to carry out. They are executed sequentially starting with the first statement, then the second, then the third and so on until the final statement has been executed. One of the simplest and most common statements is to use `System.out.println` to produce a line of output. This is another magical incantation that people tend to just memorize as if it were one word. As of this

writing, Google lists over 6,000,000 web pages that mention `System.out.println`. The key thing to know about `System.out.println` is that we use it to produce a line of output that is sent to the console window.

The simplest form of the println has nothing inside parentheses and produces a blank line of output:

```
System.out.println();
```

You need to include the parentheses even if you don't have anything to put inside them. Notice the semicolon at the end of the line. All statements in Java must be terminated with a semicolon.

More often you use println to output a line of text:

```
System.out.println("This line uses the println method.");
```

This statement commands the computer to produce the following line of output.

```
This line uses the println method.
```

Each println statement produces a different line of output. These three statements each produce a line of output (the second is blank):

```
System.out.println("This is the first line of output.");
System.out.println();
System.out.println("And this is the third, below a blank line.");
```

The following three lines of output are produced:

```
This is the first line of output.

And this is the third, below a blank line.
```

Escape Sequences

Any system that involves quoting text will lead you to certain difficult situations. For example, string literals are contained inside of quotation marks, so how could you include a quotation mark inside a string literal? String literals also aren't allowed to break across lines, so how would you include a line break inside a string literal?

The solution is that you can embed what are known as *escape sequences* in a string literal. These are two-character sequences that are used to represent special characters. They all begin with the backslash character ("\"). The following table lists some of the more common escape sequences.

Sequence	Represents
`\t`	tab character
`\n`	newline character
`\"`	quotation mark
`\\`	backslash character

Keep in mind that each of these two character sequences actually stands for just a single character. For example, if you were to execute the following statement:

```
System.out.println("What \"characters\" does this \\ produce?");
```

You would get the following output:

```
What "characters" does this \ produce?
```

The string literal in the println has three escape sequences that are each two characters long, but they each produce a single character of output.

Even though you can't have string literals that span multiple lines, you can use the `\n` escape sequence to embed newline characters in a string. This leads to the odd situation where a single println can produce more than one line of output.

For example, if you execute this statement:

```
System.out.println("This one line\nproduces 3 lines\nof output.");
```

You will get the following output:

```
This one line
produces 3 lines
of output.
```

The println itself produces one line of output and the string literal contains two newline characters in the middle that cause the line of output to be broken up into a total of three lines of output.

This is another programming habit that tends to vary according to taste. Some people (including the authors) find it hard to read string literals that have `\n` in them, but other people prefer to write fewer lines of code. Once again, you should make up your own mind about when to use the newline escape sequence.

Identifiers and Keywords

The words used to name parts of a Java program are called *identifiers*. An identifier specifies the name of a class, method, or other entity in your program.

Identifier

A name given to an entity in a program such as a class or method.

Identifiers must start with a letter and can then be followed by any number of letters and digits. The following are legal identifiers.

```
first          hiThere     numStudents     TwoBy4
```

The Java language specification defines the set of letters to include the underscore and dollar-sign characters (_ and `$`), which means that the following are legal identifiers as well.

```
two_plus_two    _count      $2donuts        MAX_COUNT
```

The following are illegal identifiers.

```
two+two         hi there    hi-There        2by4
```

Java has conventions for capitalization that are followed fairly consistently by programmers. All class names should begin with a capital letter, as with the Hello, Hello2 and Hello3 classes in the previous section. The names of methods should begin with lowercase letters, as in method main. When putting several words together, we capitalize the first letter of each word. In the next chapter we'll learn about constants, which have an even more distinct capitalization scheme, with all letters in uppercase and words separated by underscores. These different schemes might seem like tedious constraints, but using consistant capitalization in our code allows us to quickly identify what kind of code element we are looking at.

For example, suppose that you were going to put together the words "all my children" into an identifier. Depending upon what the identifier is used for, you'd turn this into:

- `AllMyChildren` for a class name (starts with a capital, capitalizes remaining words)
- `allMyChildren` for a method name (starts with a lowercase letter, capitalizes remaining words)
- `ALL_MY_CHILDREN` for a constant name (all uppercase separated by underscores; described in Chapter 2)

Java is case sensitive, so the identifiers `class`, `Class`, `CLASS` and `cLASs` are all considered different words. Keep this in mind as you read error messages from the compiler. People are good at understanding what you are saying even if you make little mistakes like changing the capitalization of a word. The Java compiler becomes hopelessly confused when you misspell even a single word.

Don't hesitate to use long identifiers. The more descriptive your names, the easier it will be for people (including yourself) to read your programs. Descriptive identifiers are worth the time they take to type. Java's String class, for example, has a method called compareToIgnoreCase.

Java has a set of predefined identifiers called *keywords* that are reserved for a particular use. As you read this book, you will learn many of these keywords and what they are used for. You can only use keywords for their intended purpose. You must be careful to avoid using these words for definitions that you make. For example, if you name a method `short` or `try`, this will cause a problem, because short and try are reserved keywords. Here is the complete list of reserved keywords.

List of Java Keywords

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	
continue	goto	package	synchronized	

A Complex Example: DrawFigures1

The println statement can be used to draw text figures as output. Consider this more complicated program example. Notice that it uses two empty println statements to produce blank lines.

```
 1  public class DrawFigures1 {
 2      public static void main(String[] args) {
 3          System.out.println("   /\\");
 4          System.out.println("  /  \\");
 5          System.out.println(" /    \\");
 6          System.out.println(" \\    /");
 7          System.out.println("  \\  /");
 8          System.out.println("   \\/");
 9          System.out.println();
10          System.out.println(" \\    /");
11          System.out.println("  \\  /");
12          System.out.println("   \\/");
13          System.out.println("   /\\");
14          System.out.println("  /  \\");
15          System.out.println(" /    \\");
16          System.out.println();
17          System.out.println("   /\\");
18          System.out.println("  /  \\");
19          System.out.println(" /    \\");
20          System.out.println("+------+");
21          System.out.println("|      |");
22          System.out.println("|      |");
23          System.out.println("+------+");
24          System.out.println("|United|");
25          System.out.println("|States|");
26          System.out.println("+------+");
27          System.out.println("|      |");
28          System.out.println("|      |");
29          System.out.println("+------+");
30          System.out.println("   /\\");
31          System.out.println("  /  \\");
32          System.out.println(" /    \\");
33      }
34  }
```

Here is the output it generates. Notice that the program has double backslash characters (`\\`) but the output has single backslash characters. This is an example of an escape sequence as described previously.

Comments and Readability

The layout of a program can enhance its readability. Java is a free-format language. This means you can put in as many or as few spaces and blank lines as you like, as long as you put at least one space or other punctuation mark between words. The following program is legal, but hard to read.

```
1  public class Ugly{public static void main(String[] args)
2  {System.out.println("How short I am!");}}
```

Here are some simple rules to follow that will make your programs more readable.

- Put class and method headers on lines by themselves.
- Put no more than one statement on each line.
- Indent your program properly. When a `{` brace appears, increase the indentation of following lines. When a `}` brace appears, reduce the indentation. Indent statements inside curly braces by a consistent number of spaces (a common choice is 4 spaces per level of indentation).
- Use blank lines to separate parts of the program (e.g., methods)

Using these rules to rewrite the program above yields the following.

```
1  public class Ugly {
2      public static void main(String[] args){
3          System.out.println("How short I am!");
4      }
5  }
```

Well-written Java programs can be quite readable, but often you will want to include some explanations that are not part of the program itself. You can annotate programs by putting notes called *comments* in them.

Comment

Text included by programmers to explain their code. Comments are ignored by the compiler.

There are two comment forms in Java. In the first form, you open the comment with a slash followed by a star and you close it by a star followed by a slash:

```
/* like this */
```

You must not put spaces between the slashes and the asterisks:

```
/ * this is bad * /
```

You can put almost any text you like, including multiple lines inside the comment.

```
/* Thaddeus Martin
   Assignment #1
   Instructor:  Professor Walingford
   Grader:      Bianca Montgomery       */
```

The only thing you aren't allowed to put inside a comment is the comment end character(s). The following is not legal.

```
/* This comment has an asterisk/slash /*/ in it
   which prematurely closes the comment.  This is bad.   */
```

Java provides a second comment form for shorter, single-line comments. You can use two slashes in a row to indicate that the rest of the current line (everything to the right of the two slashes) is a comment. For example, you can put a comment after a statement:

```
System.out.println("You win!");  // Good job!
```

Or you can create a comment on its own line:

```
// give an introduction to the user
System.out.println("Welcome to the game of blackjack.");
System.out.println();
System.out.println("Let me explain the rules.");
```

You can even create blocks of single-line comments.

```
// Thaddeus Martin
// Assignment #1
// Instructor:  Professor Walingford
// Grader:      Bianca Montgomery
```

Some people prefer to use the other comment form in a case like this when you know you will have multiple lines of text in the comment, but it is safer to use the second form because you don't have to remember to close the comment. Plus, it makes the comment stand out. This is another time where, if you are not told to use a particular comment style by an instructor or colleague, you should decide for yourself which style you prefer and use it consistently.

Don't confuse comments with the text of the println statements. The text of your comments will not be displayed as output when the program executes. The comments are to help examine and understand the program.

It is a good idea to include comments at the beginning of each class file to indicate what the class does. You might also want to include information about who you are, what course you are taking, your instructor and/or grader's name, the current date, et cetera. You should also comment each method to indicate what it does.

Commenting becomes more useful in larger and more complicated programs, as well as programs that will be viewed or modified by more than one programmer. Clear comments are extremely helpful to explain to another person, or to you yourself at a later time, what your program is doing and why it is doing it.

Java has a particular style of comments known as Javadoc comments. They have a more complex format to them but they have the advantage that you can use a program to extract the comments to make HTML files suitable for reading with a web browser. Javadoc comments are useful in more advanced programming and are not discussed here in detail.

1.3 Program Errors

In 1949 Maurice Wilkes, an early pioneer of computing, expressed a sentiment that still rings true today.

> As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.
>
> --Maurice Wilkes

You will also have to face this reality as you learn to program. You're going to make mistakes just like every other programmer in history and you're going to need strategies for eliminating those mistakes. Fortunately the computer itself can help you with some of the work.

There are three kinds of errors that you'll encounter as you write programs:

- *Syntax errors* occur when you misuse Java. They are the programming equivalent of bad grammar and are caught by the Java compiler.

- *Logic errors* occur when you write code that doesn't perform the task it is intended to perform.
- *Runtime errors* are logic errors that are so severe that Java stops your program from executing.

Syntax Errors

Human beings tend to be fairly forgiving about minor mistakes in speech. For example, we might find it odd but we generally understand Master Yoda when he says, "Unfortunate that you rushed to face him...that incomplete was your training. Not ready for the burden were you."

The Java compiler will be far less forgiving. The compiler reports syntax errors as it attempts to translate your program from Java into bytecodes, if your program breaks any of Java's grammar rules. For example, if you misplace a single semicolon in your program, you can send the compiler into a tail spin of confusion. And once confused, the compiler might get very confused. The compiler may report several error messages, depending on what it thinks is wrong with your program.

A program that generates compilation errors cannot be executed. If you submit your program to the compiler and have errors reported, you must fix the errors and resubmit the program to the compiler. You will not be able to proceed until your program is free of compilation errors.

An error can be introduced before you even start writing your program, if you choose the wrong name for its file.

Common Programming Error: File name does not match class name

As mentioned earlier, Java requires that a program's class name and file name match. For example, a program that begins with `public class Hello` must be stored in a file called `Hello.java`.

If you use the wrong file name (say, saving it as `WrongFileName.java`), you'll get an error message like this:

```
WrongFileName.java:1: class Hello is public, should be declared in a file named Hello.java
public class Hello {
       ^
1 error
```

Once your file name is correct, there are still a number of errors that may exist in your Java program. One of the most common syntax errors is to misspell a word.

Common Programming Error: Misspelled words

Java (like most programming languages) is very picky about spelling. You need to spell each word correctly, including proper capitalization. Suppose, for example, that we replace the println statement in the "hello world" program with the following:

```
System.out.pruntln("Hello, world!");
```

When we try to compile this program, it generates an error message similar to the following:

```
Hello.java:3: cannot find symbol
symbol  : method pruntln(java.lang.String)
location: class java.io.PrintStream
        System.out.pruntln("Hello, world!");
                   ^
1 error
```

The first line of this output indicates that the error occurs in the file Hello.java on line 3 and that the error is that the compiler "cannot find symbol". The second line indicates that the symbol it can't find is a method called `pruntln`. That's because there is no such method; the method is called `println`. The error message can take slightly different forms depending upon what you have misspelled. For example, if you forget to capitalize the word `System`:

```
system.out.println("Hello, world!");
```

You will get the following error message:

```
Hello.java:3: package system does not exist
        system.out.println("Hello, world!");
              ^
1 error
```

Again the first line indicates that the error occurs in line 3 of the file Hello.java. The error message is slightly different here, indicating that it can't find anything called `system`. The second and third lines of this error message include the original line of code with an arrow pointing to where the compiler got confused. The compiler errors are not always very clear, but if you pay attention to where the error is pointing, you have a pretty good sense of about where the error occurs.

Some Java editors, such as Eclipse, also underline your syntax errors for you as you write your program. This makes it easier to see the line and portion of the program that contains the error.

Once your spelling is correct, you may still have errors related to punctuation, such as missing semicolons.

Common Programming Error: Forgetting a semicolon

All Java statements end with semicolons. It's easy to forget to put a semicolon at the end of a statement, as is done in the following program:

```
1  public class MissingSemicolon {
2      public static void main(String[] args) {
3          System.out.println("A rose by any other name")
4          System.out.println("would smell as sweet");
5      }
6  }
```

The compiler produces output similar to the following:

```
MissingSemicolon.java:4: ';' expected
        System.out.println("would smell as sweet");
        ^
1 error
```

The odd thing about the compiler's output is that it's listed line 4 as the cause of the problem, not line 3 where the semicolon was actually forgotten. This is because the compiler is looking forward for a semicolon and isn't upset until it finds something that isn't a semicolon, which it does when it reaches line 4. Unfortunately in cases like this compiler error messages don't always direct you to the correct line to be fixed.

Semicolons are not the only item that you can forget when writing your program. It's also easy to forget an entire word, such as a required keyword.

Common Programming Error: Forgetting a required keyword

Another common syntax error is to forget a required keyword when typing your program, such as `static` or `class`. Double-check your programs against the examples in the textbook to make sure you haven't omitted an important keyword.

The compiler will give different error messages depending on which keyword is missing, but the messages can be hard to understand. For example, you might write a program named `Bug4` and forget the keyword `class` when writing its class header. The compiler provides the following error message:

```
Bug4.java:1: 'class' or 'interface' expected
public Bug4 {
       ^
1 error
```

However, if you forget the keyword `void` when declaring the `main` method, the compiler generates a different error message:

```
Bug5.java:2: invalid method declaration; return type required
    public static main(String[] args) {
                  ^
1 error
```

The error message the compiler gives may or may not be helpful. Many times the message will confuse you, because it will use words and terms that we haven't discussed yet. Notice that in these cases the error contains a caret marker, `^`, pointing at the position in the line where the compiler became confused. This can help you pinpoint the place where a required keyword might be missing.

If you can't understand the error message, looking at the error's line number and comparing the contents of that line with similar lines in other programs is a good place to start. Also, asking someone else such as an instructor or lab assistant to examine your program can often uncover such mistakes.

Another common syntax error is to forget to close a string literal.

Common Programming Error: Not closing a string literal or comment

Every string literal has to have an opening quote and a closing quote, but it's easy to forget the closing quotation mark. For example, you might say:

```
System.out.println("Hello, world!);
```

This produces two different error messages even though there is only one underlying error:

```
Hello.java:3: unclosed string literal
        System.out.println("Hello, world!);
                           ^
Hello.java:4: ')' expected
    }
    ^
2 errors
```

In this case, the first error message is quite clear, including an arrow pointing at the beginning of the string literal that wasn't closed. The second error message was caused by the first. Because the string literal was not closed, the compiler didn't notice the right parenthesis and semicolon that appear at the end of the line. This is an example of a single syntax error generating several different error messages.

A similar problem occurs when you forget to close a multi-line comment by writing `*/` , as is done in the following program:

```
/* This is a bad program.

public class Bad {
    public static void main(String[] args){
        System.out.println("Hi there.");
    }
} /* end of program */
```

The preceding file is not a program; it is one long comment. Because the comment on the first line is not closed, the entire program is swallowed up.

Luckily, many Java editor programs color the parts of a program to help you identify them visually. Usually if you forget to close a string literal or comment, the rest of your program will turn the wrong color, which can help you spot the mistake.

A good rule of thumb to follow is that the first error reported by the compiler is the most important one to pay attention to. The rest might be the result of that first error. Many programmers don't even bother to look at errors beyond the first. Fixing the first error and recompiling may cause the other errors to disappear.

Logic Errors (bugs)

Logic errors are also called *bugs*. Computer programmers use words like bug-ridden and buggy to describe poorly written programs. The word "bug" is an old engineering term that predates computers; early computing bugs were sometimes in hardware as well as software.

Admiral Grace Hopper, an early pioneer of computing, is largely credited with popularizing the use of the term "bug" in computer programming context. She often told the the true story of a group of programmers at Harvard University in the mid-1940s who couldn't figure out what was wrong with their programs, until they opened up the computer and found an actual moth trapped inside. The process of finding and eliminating bugs from programs is called *debugging*.

The form that a bug takes will vary. Sometimes your program will simply behave improperly. For example, it might produce the wrong output. Other times it will ask the computer to perform some task that is clearly a mistake, in which case your program will have a runtime error that stops it from executing. In this chapter, since our knowledge of Java is limited, generally the only type of logic error we will see is a mistake in program output from an incorrect println statement or method call.

1.4 Procedural Decomposition

Brian Kernighan, one of the creators of the C programming language, has said that, "Controlling complexity is the essence of computer programming." People have only a modest capacity for detail. We can't solve complex problems all at once. Instead, we structure our problem-solving by dividing the problem into manageable pieces and conquering each piece individually. We often use the term *decomposition* to describe this principle as applied to programming.

> **Decomposition**
>
> A separation into discernible parts, each of which is simpler than the whole.

With procedural programming languages like Kernighan's C, decomposition involves dividing a complex task into a set of subtasks. This is very verb or action oriented, dividing up the overall action into a series of smaller actions. This technique is called procedural decomposition.

Java was designed for a different kind of decomposition that is more noun or object oriented. Instead of thinking of the problem as a series of actions to be performed, we think of it as a collection of objects that have to interact.

As a computer scientist you should be familiar with both types of problem solving. This book begins with procedural decomposition and devotes many chapters to mastering various aspects of the procedural approach. Only after you have thoroughly practiced procedural programming will we turn our attention back to object decomposition and object oriented programming.

As an example of procedural decomposition, consider the problem of baking a cake. You can divide this problem into the following subproblems.

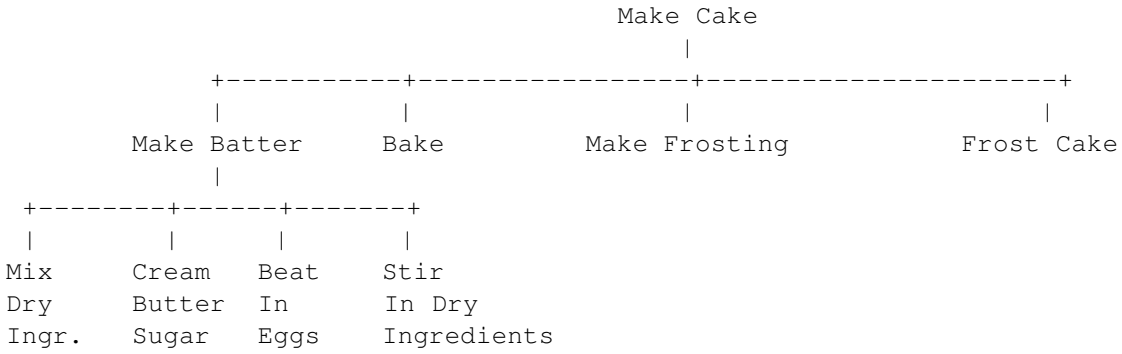
- Make the batter.
- Bake the cake.
- Make the frosting.
- Frost the cake.

Each of these four tasks has details associated with it. To make the batter, for example, you:

- Mix the dry ingredients.
- Cream the butter and the sugar.
- Beat in the eggs.
- Stir in the dry ingredients.

Thus, you divide the overall task into subtasks and further divide these subtasks into smaller subtasks. Eventually, you reach descriptions that are so simple they require no further explanation (i.e., primitives).

A diagram of this partial solution would look like this:



"Make Cake" is the highest-level operation of all. It is defined in terms of four lower-level operations called "Make Batter," "Bake," "Make Frosting," and "Frost Cake." The "Make Batter" operation is defined in terms of even lower-level operations. We use diagrams like this throughout the book. They are called structure diagrams, and are intended to show how a problem is broken down into subproblems. In this diagram, you can also tell in what order operations are performed by reading left to right. That will not be true of most structure diagrams. To determine the actual order that sub-programs are performed, you will usually have to refer to the program itself.

Two styles of approaching problems are called bottom-up and top-down. To make a cake, you might start at the top and reason, "I can make a cake by first making batter, then baking it, then making frosting, and finally putting the frosting on the cake. I could make the batter by first . . .." This is a top-down solution; it starts at the highest level and proceeds downward.

A bottom-up solution involves reasoning like, "What can I do now? I could put together dry ingredients. If I then creamed the butter and sugar, and then beat in eggs, and then mixed in dry ingredients, I'd have some batter. If I then baked that batter, I'd have an unfrosted cake. If I then . . . ." This is a bottom-up solution because it starts with low-level operations and puts them together to form higher-level operations.

One final problem solving term has to do with the process of programming. Professional programmers develop programs in stages. Instead of trying to produce a complete working program all at once, they choose some piece of the problem to implement first. Then another piece is added and another and another. The overall program is built up slowly, piece by piece. This process is known as iterative enhancement or stepwise refinement.

Iterative Enhancement

The process of producing a program in stages, adding new functionality at each stage. A key feature of each iterative step is that you can test it to make sure that piece works before moving on.

We'll now discuss a construct that will allow us to iteratively enhance our Java programs to improve their structure and reduce their redundancy: static methods.

Static Methods

Java is designed for objects and programming in Java usually involves decomposing a problem into various objects, each with methods that perform particular tasks. We will see how this works eventually, but for now, we are going to explore procedural decomposition. This will allow us to postpone some of Java's details while we learn about programming in general.

The first decomposition construct we will study is known as a static method.

Static Method

A block of Java statements which is given a name. Static methods are units of procedural decomposition. We typically break a class into several static methods, each of which solves some piece of the overall problem.

Consider the following program, which draws two text boxes on the console:

```
 1  public class DrawBoxes {
 2      public static void main(String[] args) {
 3          System.out.println("+------+");
 4          System.out.println("|      |");
 5          System.out.println("|      |");
 6          System.out.println("+------+");
 7          System.out.println();
 8          System.out.println("+------+");
 9          System.out.println("|      |");
10          System.out.println("|      |");
11          System.out.println("+------+");
12      }
13  }
```

The program works correctly, but the four lines used to draw the box are repeated twice. This redundancy is undesirable for several reasons. We might wish to change the appearance of the boxes, in which case we'd have to change both places in the program. Also, we might wish to draw additional boxes, which would require us to type additional copies of (or copy-and-paste) the redundant lines.

A preferable solution would be to create a Java command that specifies how to draw a box, and then to execute that command twice. Java doesn't have a "draw a box" command, but we can create one. To create such a command, we write a line that gives a name to the command, and then between a pair of `{` and `}` braces we place the statements to be executed by that command. Such a named command is called a *static method*. For example, here is a static method to draw a text box:

```
public static void drawBox() {
    System.out.println("+------+");
    System.out.println("|      |");
    System.out.println("|      |");
    System.out.println("+------+");
}
```

You have already seen a static method in the programs we have written called main. Recall that the main method has the following form:

```
public static void main(String[] args) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

The static methods we'll write have a similar structure:

```
public static void <name>() {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

The first line is known as the method header. For now, you will not fully understand what each word of this header means in Java; the most important thing to remember is that you'll write `public static void`, followed by the name you wish to give the method, followed by two parentheses, `()` . For the sake of completeness, we'll briefly discuss what the words in the header mean.

- The keyword `public` indicates that this method is available to be used by all parts of your program. All methods we write will be `public`.
- The keyword `static` indicates that this is a static (procedural style, not object-oriented) method. For now, all methods we write will be `static`, until we learn about writing objects in Chapter 8.
- The word `void` indicates that this method executes statements but does not compute and emit any value. (Other methods we'll see later compute and *return* values outward.)
- The word `drawBox` is the name of the method.
- The empty parentheses specify a list (in this case, an empty list) of values that are sent inward to your method as input; such values are called *parameters* and will not be used by our methods until Chapter 3.

It can be cumbersome or confusing to have to include the keyword `static` for each of these methods. Other Java textbooks often do not discuss static methods as early as we will here; instead, other techniques are shown for decomposing problems. But even though static methods require a bit of work to create, they are powerful and useful tools for improving basic Java programs.

After the header you see a series of println statements that make up the body of this static method. As in method main, the statements of this method are executed in order from first to last when the method executes.

By defining the method drawBox, we have given a simple name to this sequence of println statements. It's as if we said to the Java compiler, "Whenever I tell you to 'drawBox', I really mean that you should execute those println statements in the drawBox method." But the command won't actually be executed unless our main method explicitly says that it wants to do so. The act of executing a static method is called a *method call*.

Method Call

A command to execute another method, which causes all of the statements inside that method to be executed.

For the drawBox method, a program would call it by saying:

```
drawBox();
```

Since we want to execute the drawBox command twice (we want to draw two boxes), the main method should contain two calls to the drawBox method. The following program uses the drawBox method to produce the same output as the original DrawBoxes program.

```
 1  public class DrawBoxes2 {
 2      public static void main(String[] args) {
 3          drawBox();
 4          System.out.println();
 5          drawBox();
 6      }
 7
 8      public static void drawBox() {
 9          System.out.println("+------+");
10          System.out.println("|      |");
11          System.out.println("|      |");
12          System.out.println("+------+");
13      }
14  }
```

Flow of Control

The most confusing thing about static methods is that programs with static methods do not execute sequentially from the top to the bottom. Rather, each time the program encounters a static method call, the execution of the program "jumps" to that static method and executes each statement in that method in order, then "jumps" back to the point where the call began and resumes executing. The order in which the statements of a program are executed is called the program's *flow of control*.

Flow of Control

The order in which the statements of a Java program are executed.

Let's look at the control flow of the DrawBoxes2 program shown previously. It has two methods. The first method is the familiar method main and the second is method drawBox. As with any Java program, execution starts with the main method.

```
public static void main(String[] args) {
    drawBox();
    System.out.println();
    drawBox();
}
```

We execute each of the statements listed in the main method from first to last. Then we're done with execution. So, in a sense, the execution of this program is sequential as always.

But this main method includes two different calls on the drawBox method. This program is going to do three different things: execute drawBox, execute a println, execute drawBox again.

It is often useful to *trace* the flow of control. The following diagram captures the idea that we are executing the main method, which involves three different steps, two of which involve executing the drawBox method.

```
execute main method:
        execute drawBox method
        System.out.println();
        execute drawBox method
```

But what happens when we ask Java to execute the drawBox method? We transfer control to the method and execute each of its statements and then come back to where we left off. So we can expand this diagram to show what happens when we make the two calls on the drawBox method:

```
execute main method:
        execute drawBox method:
                System.out.println("+------+");
                System.out.println("|      |");
                System.out.println("|      |");
                System.out.println("+------+");
        System.out.println();
        execute drawBox method:
                System.out.println("+------+");
                System.out.println("|      |");
                System.out.println("|      |");
                System.out.println("+------+");
```

The transfer of control to the drawBox method causes its four println methods to be executed first. Then we return to the main method and execute its println method. Then we call drawBox again and execute its four println statements a second time.

The net effect is that we could instead have executed the following main method:

```
public static void main(String[] args) {
    System.out.println("+------+");
    System.out.println("|      |");
    System.out.println("|      |");
    System.out.println("+------+");
    System.out.println();
    System.out.println("+------+");
    System.out.println("|      |");
    System.out.println("|      |");
    System.out.println("+------+");
}
```

This version is simpler in terms of its flow of control, but the other version avoids the redundancy of having the same println statements appear multiple times. It also gives a better sense of the structure of the solution. In the original version it is clear that there is a subtask that we have called drawBox that is being performed twice. In addition, it would be easier to add a third box to the program that uses the drawBox method, since this would simply be a third method call in main, rather than having to add four new println statements.

Java allows you to define methods in any order you like. It is a common convention to put the main method either as the first or last method in the class. In this textbook we will generally put main first, but the program would behave the same if we switch the order. Method main is always the starting point for program execution and from that starting point we can determine the order in which other methods are called. For example, the following modified program behaves identically to the previous DrawBoxes2 program:

```
1  public class DrawBoxes3 {
2      public static void drawBox() {
3          System.out.println("+------+");
4          System.out.println("|      |");
5          System.out.println("|      |");
6          System.out.println("+------+");
7      }
8
9      public static void main(String[] args) {
10         drawBox();
11         System.out.println();
12         drawBox();
13     }
14 }
```

Methods That Call Other Methods

The main method is not the only place where you can call another method. In fact, any method may call any other method. As a result, the flow of control can get quite complicated. Consider, for example, the following rather strange program. We use nonsense words "foo", "bar", "baz" and "mumble" on purpose because the program is not intended to make sense.

```
1  public class FooBarBazMumble {
2      public static void main(String[] args) {
3          foo();
4          bar();
5      }
6
7      public static void foo() {
8          System.out.println("foo");
9          mumble();
10         System.out.println();
11     }
12
13     public static void bar() {
14         System.out.println("bar");
15         baz();
16     }
17
18     public static void baz() {
19         System.out.println("baz");
20         mumble();
21     }
22
23     public static void mumble() {
24         System.out.println("mumble");
25     }
26 }
```

You can't tell easily what output this program produces. Let's again trace in detail what the program is doing. Remember that Java always begins with the method called `main`. In this program, the `main` method does two things: it calls the `foo` method and it calls the `bar` method:

```
execute main method:
        execute foo method
        execute bar method
```

Each of these two method calls is going to expand into more statements. Let's first expand the call on the `foo` method. How do we execute the `foo` method? We transfer control to the method and execute each of its statements. It has three different statements:

```
execute main method:
        execute foo method:
                System.out.println("foo");
                execute mumble method
                System.out.println();
        execute bar method
```

This helps to make our trace more complete, but notice that `foo` is calling the `mumble` method, so we have to expand that as well. What happens when we execute the `mumble` method that `foo` calls? We transfer control to it and execute its statements. The `mumble` method has a single println statement:

```
execute main method:
        execute foo method:
                System.out.println("foo");
                execute mumble method:
                        System.out.println("mumble");
                System.out.println();
        execute bar method
```

At this point we have finished expanding the details of the call on the `foo` method from `main`, but we haven't yet looked at what is involved in executing the `bar` method. Control would transfer to the method and we execute each of its statements. The `bar` method has two statements:

```
execute main method:
        execute foo method:
                System.out.println("foo");
                execute mumble method:
                        System.out.println("mumble");
                System.out.println();
        execute bar method:
                System.out.println("bar");
                execute baz method
```

We're getting closer, but we have to figure out what happens when we execute the `baz` method. It has two steps as well:

```
execute main method:
        execute foo method:
            System.out.println("foo");
            execute mumble method:
                    System.out.println("mumble");
            System.out.println();
        execute bar method:
            System.out.println("bar");
            execute baz method:
                    System.out.println("baz");
                    execute mumble method
```

This leads to yet another method call that we need to expand. The call on `mumble` causes its single println to be executed:

```
execute main method:
        execute foo method:
            System.out.println("foo");
            execute mumble method:
                    System.out.println("mumble");
            System.out.println();
        execute bar method:
            System.out.println("bar");
            execute baz method:
                    System.out.println("baz");
                    execute mumble method:
                            System.out.println("mumble");
```

Finally we have finished tracing the executing of this program. It should make sense, then, that the program produces the following output:

```
foo
mumble

bar
baz
mumble
```

We will see a much more useful example of methods calling methods when we go through the case study at the end of the chapter.

Did You Know: Hacker's Dictionary

Computer scientists and computer programmers use a lot of jargon that can be confusing to novices. A group of software professionals spearheaded by Eric Raymond collected together many of the jargon terms in a book called *The New Hacker's Dictionary*. You can buy the book but you can also browse it online at Eric's website:

`http://catb.org/~esr/jargon/html/frames.html`

For example, if you look up *foo*, you'll find a definition that says it is "Used very generally as a sample name for absolutely anything, esp. programs and files." In other words, when we find ourselves looking for a nonsense word, we use "foo".

The *Hacker's Dictionary* contains a great deal of historical information about the origins of jargon terms. The entry for *foo* includes a lengthy discussion of the combined term *foobar* and how it came into common usage among engineers.

If you want to get a flavor of what is there, check out the entries for *bug*, *hacker* and *bogosity* and *bogo-sort*.

An Example Runtime Error

Runtime errors occur when a bug causes your program to be unable to continue executing. What could cause such a thing to happen? One example is if you asked the computer to calculate an invalid value, such as 1 / 0. Another example would be if your program tries to read data from a file that does not exist.

We don't know how to compute values or read files yet, but there is a way we can accidentally cause a runtime error. The way to do this is to write a static method that calls itself. If you do this, your program will not stop running, because the method will keep calling itself indefinitely, until the computer runs out of memory. When this happens, the program prints a large number of lines of output but eventually stops executing with an error message called a StackOverflowError.

```
1  public class Infinite {
2      public static void main(String[] args) {
3          oops();
4      }
5
6      public static void oops() {
7          System.out.println("Make it stop!");
8          oops();
9      }
10 }
```

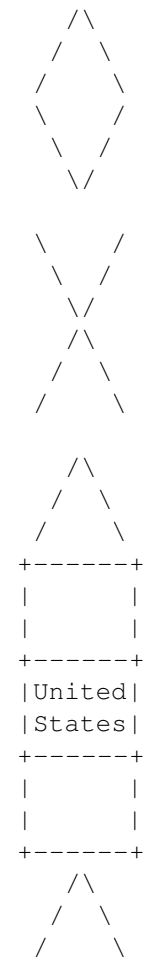
This ill-fated program produces the following output (with large groups of identical lines represented by ...):

```
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
...

Make it stop!
Exception in thread "main" java.lang.StackOverflowError
        at sun.nio.cs.SingleByteEncoder.encodeArrayLoop(Unknown Source)
        at sun.nio.cs.SingleByteEncoder.encodeLoop(Unknown Source)
        at java.nio.charset.CharsetEncoder.encode(Unknown Source)
        at sun.nio.cs.StreamEncoder$CharsetSE.implWrite(Unknown Source)
        at sun.nio.cs.StreamEncoder.write(Unknown Source)
        at java.io.OutputStreamWriter.write(Unknown Source)
        at java.io.BufferedWriter.flushBuffer(Unknown Source)
        at java.io.PrintStream.newLine(Unknown Source)
        at java.io.PrintStream.println(Unknown Source)
        at Infinite.oops(Infinite.java:7)
        at Infinite.oops(Infinite.java:8)
        at Infinite.oops(Infinite.java:8)
        at Infinite.oops(Infinite.java:8)
        at ...
```

Runtime errors are something we'll unfortunately have to live with as we learn to program. We will have to carefully ensure that our programs not only compile successfully, but that they do not contain any bugs that will cause a runtime error. The most common way to catch and fix runtime errors is to run the program several times to test its behavior.

1.5 Case Study: DrawFigures

Earlier in the chapter we saw a program called DrawFigures1 that produced the following output.



It did so with a long sequence of println statements in the main method. In this section we'll improve the program using static methods for procedural decomposition to capture structure and eliminate redundancy.

Structured Version

If we look closely at this output, we can see that there is a structure to it that would be desirable to capture in our program structure. The output is divided into three subfigures.

We can better indicate the structure of the program by dividing it into static methods. Since there are three subfigures, we'll create three methods, one for each subfigure. The following program produces the same output as DrawFigures1.

```
1  public class DrawFigures2 {
2      public static void main(String[] args) {
3          drawDiamond();
4          drawX();
5          drawRocket();
6      }
7
8      public static void drawDiamond() {
9          System.out.println("   /\\");
10         System.out.println("  /  \\");
11         System.out.println(" /    \\");
12         System.out.println(" \\    /");
13         System.out.println("  \\  /");
14         System.out.println("   \\/");
15         System.out.println();
16     }
17
18     public static void drawX() {
19         System.out.println(" \\    /");
20         System.out.println("  \\  /");
21         System.out.println("   \\/");
22         System.out.println("   /\\");
23         System.out.println("  /  \\");
24         System.out.println(" /    \\");
25         System.out.println();
26     }
27
28     public static void drawRocket() {
29         System.out.println("   /\\");
30         System.out.println("  /  \\");
31         System.out.println(" /    \\");
32         System.out.println("+------+");
33         System.out.println("|      |");
34         System.out.println("|      |");
35         System.out.println("+------+");
36         System.out.println("|United|");
37         System.out.println("|States|");
38         System.out.println("+------+");
39         System.out.println("|      |");
40         System.out.println("|      |");
41         System.out.println("+------+");
42         System.out.println("   /\\");
43         System.out.println("  /  \\");
44         System.out.println(" /    \\");
45     }
46 }
```

The program appears in a class called DrawFigures2 and has four static methods (including main) defined within it. The first static method is the usual method main, which calls three methods. These three methods called by main appear next.

Final Version without Redundancy

Our program can be further improved. Each of the three subfigures has individual elements, and some of those elements appear in more than one of the three subfigures. The following is one redundant group of lines that is printed several times by the program.

```
  /\
 /  \
/    \
```

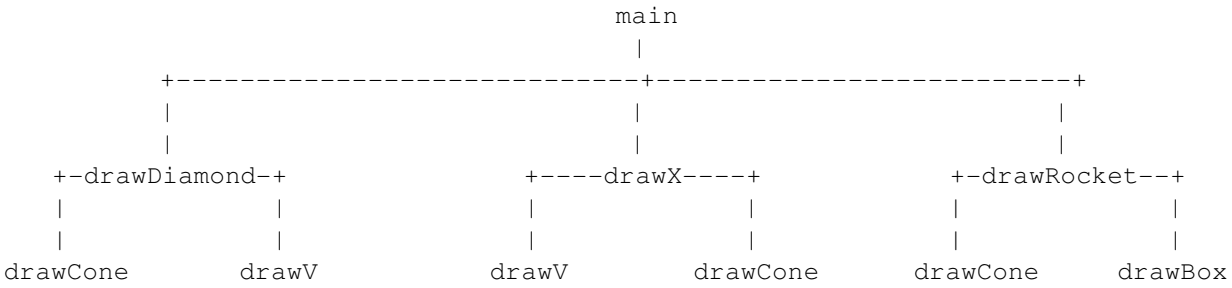
A better version of the preceding program adds an additional method for each redundant section of output. The redundant sections are the upward diamond shape, the downward "V" shape, and the box used in the rocket. Here is the improved program:

```
1  public class DrawFigures3 {
2      public static void main(String[] args) {
3          drawDiamond();
4          drawX();
5          drawRocket();
6      }
7
8      public static void drawDiamond() {
9          drawCone();
10         drawV();
11         System.out.println();
12     }
13
14     public static void drawX() {
15         drawV();
16         drawCone();
17         System.out.println();
18     }
19
20     public static void drawRocket() {
21         drawCone();
22         drawBox();
23         System.out.println("|United|");
24         System.out.println("|States|");
25         drawBox();
26         drawCone();
27         System.out.println();
28     }
29
30     public static void drawBox() {
31         System.out.println("+------+");
32         System.out.println("|      |");
33         System.out.println("|      |");
34         System.out.println("+------+");
35     }
36
37     public static void drawCone() {
38         System.out.println("   /\\");
39         System.out.println("  /  \\");
40         System.out.println(" /    \\");
41     }
42
43     public static void drawV() {
44         System.out.println(" \\    /");
45         System.out.println("  \\  /");
46         System.out.println("   \\/");
47     }
48 }
```

This program now called DrawFigures3 has seven static methods (including main) defined within it. The first static method is the usual method main, which calls three methods. These three methods in turn call three other methods that appear next.

Analysis of Flow of Execution

Here is a structure diagram that shows which static methods main calls and which static methods are called by each of them:



As you can see, this program has three levels of structure, two levels of decomposition. The overall task is split into three subtasks. And each of those subtasks has two subtasks.

A program with methods has a more complex flow of control than one without them but the rules are still fairly simple. Remember that when a method is called, the computer executes the statements in the body of that method. Then, control proceeds to the statement after the method call. Also remember that we always start with the main method, executing its statements from first to last.

So in executing program DrawFigures3, we first execute its main method. That, in turn, first executes the body of method drawDiamond. Then drawDiamond executes methods drawCone and drawV (in that order). When drawDiamond finishes executing, control shifts to the next statement in the body of the main program, the call on method drawX.

A complete breakdown of the flow of control from static method to static method in DrawFigures3 follows.

```
 1st        main
 2nd            drawDiamond
 3rd                drawCone
 4th                drawV
 5th            drawX
 6th                drawV
 7th                drawCone
 8th            drawRocket
 9th                drawCone
10th                drawBox
11th                drawBox
12th                drawCone
```

Recall that the order in which you define methods does not have to parallel the order in which they are executed. The order of execution is determined by the body of method main and by the bodies of methods called from main. A static method declaration is like a dictionary entry--it defines a

word, but it does not specify how the word will be used. The body of this main says to first execute drawDiamond, then drawX, then drawRocket. This is the order of execution, regardless of the order they were defined in.

Java allows you to define methods in any order you like. We started with main at the top and worked our way down to lower and lower level methods. This is a popular approach to take, but many people prefer the opposite, having the low level methods first and having main at the end. Java doesn't care what order you use, so you can decide for yourself and do what you think is best. Consistency is important, so that you can find a method easily later in a larger program.

It is important to note that the programs DrawFigure1, DrawFigure2, and DrawFigure3 produce exactly the same output to the console. While DrawFigure1 may be the easiest program for a novice to read, DrawFigure2 and particularly DrawFigure3 have many other advantages over it. For one, a well-structured solution is easier to comprehend, and the methods themselves become a means of explaining a program. Also, programs with methods are more flexible, more easily adapted to a similar but different task. You can take the seven methods defined in DrawFigures3 and write the following new program to produce a larger and more complex output. Building static methods to create new commands increases your flexibility without adding unnecessary complication. For example, we could replace our main method with a version that calls our methods in the following new order. What output would it produce?

```
public static void main(String[] args) {
    drawCone();
    drawCone();
    drawRocket();
    drawX();
    drawRocket();
    drawDiamond();
    drawBox();
    drawDiamond();
    drawX();
    drawRocket();
}
```

Chapter Summary

- Computers execute sets of instructions called programs. Computers store information internally as sequences of 0s and 1s (binary numbers).
- Programming and computer science deal with algorithms, which are step-by-step descriptions for solving problems.
- Programs are translated from text into computer instructions by another program called a compiler. Java's compiler turns Java programs into a special format called bytecode, which is executed using a special program called the Java runtime system.
- Java is a modern object-oriented programming language developed by Sun Microsystems that has a large set of libraries you can use to build complex programs.

- Java programmers typically complete their work using an editor called an Integrated Development Environment (IDE). The commands may vary from environment to environment, but always involve the same three step process:

    1. type in a program as a Java class
    2. compile the program file
    3. run the compiled version of the program

- Java uses a command called `System.out.println` to display text on the console screen.

- Written words in a program can take different meanings. Keywords are special words that are part of the language. Identifiers are words defined by the programmer to give names to entities in the program. Words can also be put into strings, which are pieces of text that can be printed to the console.

- Java programs that use proper spacing and layout are more readable to programmers. Readability is also improved by writing notes called comments inside the program.

- The Java language has a syntax, or a legal set of commands that can be used. A Java program that does not follow the proper syntax will not compile. A program that does compile but is written incorrectly may still contain errors called exceptions that occur when the program runs. A third kind of errors are logic or intent errors, when the program runs but does not do what the programmer intended.

- Commands in programs are called statements. A class can group statements into larger commands called static methods. Static methods help the programmer group code into reusable pieces. An important static method that must be part of every program is called `main`.

- Complex programming tasks should be broken down into the major tasks the computer must perform. This process is called procedural decomposition. Correct use of static methods aids procedural decomposition.

Self-Check Problems

Section 1.1: Basic Computing Concepts

1. Why do computers use binary numbers?

2. Convert each of the following decimal numbers into its equivalent binary number:
    - 6
    - 44
    - 72
    - 131

3. What is the decimal equivalent of each of the following binary numbers?
    - 100
    - 1011
    - 101010
    - 1001110

4. In your own words, describe an algorithm for baking cookies. Assume that you have a large number of hungry friends, so you'll want to produce several batches of them!

5. What is the difference between the file `MyProgram.java` and the file `MyProgram.class`?

Section 1.2: And Now--Java

6. Which of the following can be used in a Java program as identifiers?

```
println          first-name    AnnualSalary    "hello"      ABC
42isTheAnswer    for           sum_of_data     _average     B4
```

7. What is the output produced from the following statements?

```
System.out.println("\"Quotes\"");
System.out.println("Slashes \\//");
System.out.println("How '\"confounding' \"\\\" it is!");
```

8. What is the output produced from the following statements?

```
System.out.println("Shaq is 7\'1\"");
System.out.println("The string \"\" is an empty message.");
System.out.println("\\'\"\"");
```

9. What is the output produced from the following statements?

```
System.out.println("Dear \"DoubleSlash\" magazine,");
System.out.println("\tYour publication confuses me.  Is it a");
System.out.println("\\\\ slash or a //// slash that I should use?");
System.out.println("\nSincerely,");
System.out.println("Susan \"Suzy\" Smith");
```

10. What series of `println` statements would produce the following output?

```
"Several slashes are sometimes seen,"
said Sally.  "I've said so myself."  See?
\ / \\ // \\\ ///
```

11. What series of `println` statements would produce the following output?

```
This is a test of your
knowledge of "quotes" used
in 'string literals.'

You're bound to "get it right"
if you read the section on
''quotes.''
```

Section 1.3: Program Errors

12. Name the three errors in the following program.

```
1  public MyProgram {
2      public static void main(String[] args) {
3          System.out.println("This is a test")
4          System.out.Println("of the emergency broadcast system.");
5      }
6  }
```

13. Name the four errors in the following program.

```
1  public class SecretMessage {
2      public static main(string[] args) {
3          System.out.println("Speak friend");
4          System.out.println("and enter);
5
6  }
```

14. Name the three errors in the following program.

```
1  public class FamousSpeech
2      public static void main(String[]) {
3          System.out.println("Four score and seven years ago,");
4          System.out.println("our fathers brought forth on this continent");
5          System.out.println("a new nation");
6          System.out.println("conceived in liberty,");
7          System.out.println("and dedicated to the");      /* this part should
8          System.out.println("proposition that all";          really say,
9          System.out.println("men are created equal");        "all PEOPLE"! */
10     }
11 }
```

Section 1.4: Procedural Decomposition

15. What is the output of the following program? You may wish to draw a structure diagram first.

```
 1  public class Strange {
 2      public static void first() {
 3          System.out.println("Inside first method");
 4      }
 5
 6      public static void second() {
 7          System.out.println("Inside second method");
 8          first();
 9      }
10
11      public static void third() {
12          System.out.println("Inside third method");
13          first();
14          second();
15      }
16
17      public static void main(String[] args) {
18          first();
19          third();
20          second();
21          third();
22      }
23  }
```

16. What would have been the output of the preceding program if the `third` method had contained the following statements?

```
public static void third() {
    first();
    second();
    System.out.println("Inside third method");
}
```

17. What would have been the output of the `Strange` program if the `main` method had contained the following statements? (Use the original version of `third`, not the modified version from the most recent exercise.)

```
public static void main(String[] args) {
    second();
    first();
    second();
    third();
}
```

18. What is the output of the following program? You may wish to draw a structure diagram first.

```
 1  public class Confusing {
 2      public static void method2() {
 3          method1();
 4          System.out.println("I am method 2.");
 5      }
 6
 7      public static void method3() {
 8          method2();
 9          System.out.println("I am method 3.");
10          method1();
11      }
12
13      public static void method1() {
14          System.out.println("I am method 1.");
15      }
16
17      public static void main(String[] args) {
18          method1();
19          method3();
20          method2();
21          method3();
22      }
23  }
```

19. What would have been the output of the preceding program if the `method3` method had contained the following statements?

```
public static void method3() {
    method1();
    method2();
    System.out.println("I am method 3.");
}
```

20. What would have been the output of the `Confusing` program if the `main` method had contained the following statements? (Use the original version of `method3`, not the modified version from the most recent exercise.)

```
public static void main(String[] args) {
    method2();
    method1();
    method3();
    method2();
}
```

21. The following program contains at least 10 syntax errors. What are they?

```
 1  public class LotsOf Errors {
 2      public static main(String args) {
 3          System.println("Hello, world!);
 4          message()
 5      }
 6
 7      public static void message {
 8          System.out println("This program surely cannot";
 9          System.out.println("have any so-called "errors" in it");
10      }
```

22. Consider the following program, saved into a file named Example.java:

```
1  public class Example {
2      public static void displayRule() {
3          System.out println("The first rule ");
4          System.out.println("of Java Club is,");
5          System.out.println();
6          System.out.println("you do not talk about Java Club.");
7      }
8
9      public static void main(String[] args) {
10         System.out.println("The rules of Java Club.");
11         displayRule();
12         displayRule();
13     }
14 }
```

What would happen if each of the following changes was made to the Example program? Treat each change independently of the others. For example: no effect, syntax error, or different program output.

- change line 1 to: `public class Demonstration`
- change line 9 to: `public static void MAIN(String[] args) {`
- insert a new line after line 11 that reads: `System.out.println();`
- change line 2 to: `public static void printMessage() {`
- change line 2 to: `public static void showMessage() {`
  and change lines 11 and 12 to: `showMessage();`
- replace lines 3-4 with: `System.out println("The first rule of Java Club is,");`

23. The following program is legal under Java's syntax rules, but it is difficult to read because of its layout and lack of comments. Reformat it using the rules given in this chapter, and add a comment header at the top of the program.

```
1  public class GiveAdvice{ public static
2  void main(String[  ]args){ System.out.println (
3  "Programs can be easy or difficult");  System.out.println(
4  "to read, depending upon their format."
5  );  System.out.println()
6  ;  System.out.println("Everyone, including yourself, will be");System.out.println
7  (
8        "happier if you choose to format your");System.out.println("Programs."); }
9        }
```

24. The following program is legal under Java's syntax rules, but it is difficult to read because of its layout and lack of comments. Reformat it using the rules given in this chapter, and add a comment header at the top of the program.

```
1  public
2  class Messy{public
3  static void main(String[]args){message (   )
4    ;System.out.println()   ; message ( );}   public static void
5  message() { System.out.println(
6      "I really wish that"
7      );System.out.println
8  ("I had formatted my source")
9  ;System.out.println("code correctly!");}}
```

Exercises

1. Write a complete Java program that prints the following output:

```
//////////////////////
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
```

2. Write a complete Java program that prints the following output:

```
  \/
 \\//
\\\//
///\\\
 //\\
  /\
```

3. Write a complete Java program that prints the following output:

```
A well-formed Java program has a main
method with { and } braces.

A System.out.println statement has ( and )
and usually a String that starts and ends
with a " character.  (But we type \" instead!)
```

4. Write a complete Java program that prints the following output:

```
What is the difference between
a ' and a "?  Or between a " and a \"?

One is what we see when we're typing our program.
The other is what appears on the "console."
```

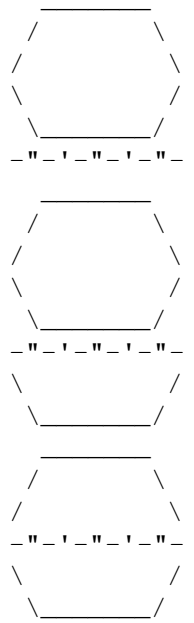
5. Write a complete Java program that prints the following output. Use at least one static method besides `main` to help you.

```
//////////////////////
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
```

6. Write a program that displays the following output:

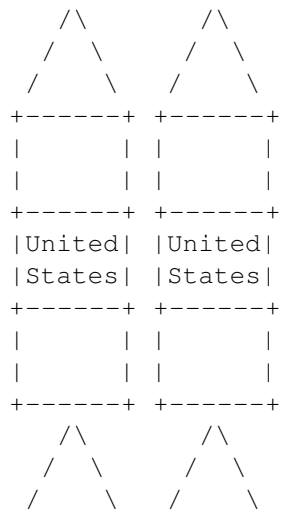
```
   _______
  /       \
 /         \
-"-'-"-'-"-
 \         /
  \_______/
```

7. Modify the program from the previous exercise so that it displays the following output. Use static methods as appropriate.



8. Write a Java program that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

   Note that there are two rocket ships next to each other. What redundancy can you eliminate using static methods? What redundancy cannot be eliminated?



9. Write a Java program that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

```
*****
*****
 * *
  *
 * *

*****
*****
 * *
  *
 * *
*****
*****

  *
  *
  *
*****
*****
 * *
  *
 * *
```

10. Write a program that prints the following line of output 1000 times:

    ```
    All work and no play makes Jack a dull boy.
    ```

    You should not write a program whose source code is 1000 lines long; use methods to shorten the program. What is the shortest program you can write that will produce the 1000 lines of output, using only the material from this chapter?

Programming Projects

1. Write a program to spell out MISSISSIPPI using block letters like the following (one per line):

    ```
    M     M        IIIII          SSSSS          PPPPPP
    MM   MM          I          S     S         P     P
    M M M M          I          S               P     P
    M  M  M          I           SSSSS          PPPPPP
    M     M          I                S         P
    M     M          I          S     S         P
    M     M        IIIII          SSSSS          P
    ```

2. Sometimes we want to write similar letters to different people. For example, you might write to your parents to tell them about your classes, your friends, and to ask for money; you might write to a friend about your love life, your classes, and your hobbies; and you might write to your brother about your hobbies, to ask for money, and your friends. Write a program that prints similar letters such as these to three people of your choice. Your letters should have at least one common paragraph. Your main program should have three method calls: one for each of the people to whom you are writing. Try to isolate repeated tasks into methods.

3. Write a program that produces as output the lyrics of the song, "There Was An Old Lady." Use methods for each verse and the refrain. Here are the song's complete lyrics:

```
There was an old lady who swallowed a fly.
I don't know why she swallowed that fly,
Perhaps she'll die.

There was an old lady who swallowed a spider,
That wriggled and iggled and jiggled inside her.
She swallowed the spider to catch the fly,
I don't know why she swallowed that fly,
Perhaps she'll die.

There was an old lady who swallowed a bird,
How absurd to swallow a bird.
She swallowed the bird to catch the spider,
She swallowed the spider to catch the fly,
I don't know why she swallowed that fly,
Perhaps she'll die.

There was an old lady who swallowed a cat,
Imagine that to swallow a cat.
She swallowed the cat to catch the bird,
She swallowed the bird to catch the spider,
She swallowed the spider to catch the fly,
I don't know why she swallowed that fly,
Perhaps she'll die.

There was an old lady who swallowed a dog,
What a hog to swallow a dog.
She swallowed the dog to catch the cat,
She swallowed the cat to catch the bird,
She swallowed the bird to catch the spider,
She swallowed the spider to catch the fly,
I don't know why she swallowed that fly,
Perhaps she'll die.

There was an old lady who swallowed a horse,
She died of course.
```

4. Write a program that produces as output the words of "The Twelve Days of Christmas." (Static methods simplify this task.) Here are the first two verses and the last verse of the song:

```
On the first day of Christmas,
my true love sent to me
a partridge in a pear tree.

On the second day of Christmas,
my true love sent to me
two turtle doves, and
a partridge in a pear tree.

...

On the twelfth day of Christmas,
my true love sent to me
Twelve drummers drumming,
eleven pipers piping,
ten lords a-leaping,
nine ladies dancing,
eight maids a-milking,
seven swans a-swimming,
six geese a-laying,
five gold rings;
four calling birds,
three French hens,
two turtle doves, and
a partridge in a pear tree.
```

Stuart Reges
Marty Stepp

Chapter 2
Primitive Data and Definite Loops

Copyright © 2006 by Stuart Reges and Marty Stepp

- 2.1 Basic Data Concepts
  - Primitive Types
  - Expressions
  - Literals
  - Arithmetic Operators
  - Precedence
  - Mixing Types and Casting
- 2.2 Variables
  - Assignment/Declaration Variations
  - String Concatenation
  - Increment/Decrement Operators
  - Variables and Mixing Types
- 2.3 The for Loop
  - Tracing for Loops
  - print Versus println
  - Nested for Loops
- 2.4 Managing Complexity
  - Scope
  - Pseudocode
  - A Decrementing for Loop
  - Class Constants
- 2.5 Case Study: A Complex Figure
  - Problem Decomposition and Pseudocode
  - Line Pattern Table
  - Initial Structured Version
  - Adding a Class Constant
  - Further Variations

Introduction

Now that you know something about the basic structure of Java programs, you are ready to learn how to solve problems that are fairly complex. You will still be restricted to programs that produce output, but we will begin to explore some of the aspects of programming that require problem solving skills.

The first half of the chapter fills in two important areas. First, it examines expressions: how to express simple computations in Java, particularly those involving numeric data. Second, it discusses program elements called variables that can change in value as the program executes.

The second half of the chapter introduces your first control structure: the for loop. You use this structure to repeat actions in a program. This is useful whenever you find a pattern in a complex task like the creation of a complex figure, because you can use a for loop to repeat an action that creates a particular pattern. The challenge is finding each pattern and figuring out what repeated actions will reproduce it. In exploring this, we will examine a variation of println known as print that allows us to break up a complex line of output into several pieces of output.

The for loop is a flexible control structure that can be used for many tasks, but in this chapter we use it for *definite loops* where you know exactly how many times you want to perform a particular task. In Chapter 5 we will discuss how to write indefinite loops where you don't know in advance how many times to perform a task.

2.1 Basic Data Concepts

Programs manipulate information and information comes in many forms. Java is a *strongly typed* language, which means that Java requires us to be explicit about what kind of information we intend to manipulate. Everything that we manipulate in a Java program will be of a certain type and we will constantly find ourselves telling Java what types of data we intend to use.

A decision was made early in the design of Java to have two different kinds of data: primitive data and objects. The designers admit that this decision was made purely on the basis of performance to make Java programs run faster. Their decision is unfortunate because it means that we have to learn two sets of rules about how data works. But this is one of those times when you simply have to pay the price if you want to use an industrial strength programming language. To make our lives a little easier, we will study the primitive data types first in this chapter and then turn our attention to objects in the next chapter.

Primitive Types

There are eight primitive data types in Java of which four are considered fundamental. The other four types are variations that exist for programs that have special requirements. The four types we will explore are listed in the following table.

Commonly Used Primitive Types in Java

Type	Description	Examples
`int`	integers (whole numbers)	`42`, `-3`, `18`, `20493`, `0`
`double`	real numbers	`7.35`, `14.9`, `-19.83423`
`char`	single characters	`'a'`, `'X'`, `'!'`
`boolean`	logical values	`true`, `false`

The type names (int, double, char and boolean) are Java keywords that we will use in our programs to let the compiler know that we intend to use that type of data.

It may seem odd to have one type for integers and another type for real numbers. Isn't every integer a real number? The answer is yes, but these are fundamentally different types of numbers. The difference is so great that we make this distinction even in English. We don't ask, "How much sisters do you have?" or "How many do you weigh?" We realize that sisters come in discrete integer quantities (0 sisters, 1 sister, 2 sisters, 3 sisters, and so on) and we use the word "many" for integer quantities ("How many sisters do you have?"). Similarly, we realize that weight can vary by tiny amounts (175 pounds versus 175.5 pounds versus 175.25 pounds and so on) and we use the word "much" for these real-valued quantities ("How much do you weigh?").

In programming this distinction is even more important because integers and reals are represented in a different way in the computer's memory. Integers are stored exactly while reals are stored as approximations with a limited number of digits of accuracy. We will see that storing values as approximations can lead to round-off errors when you use real values.

The name "double" for real values isn't very clear. It's an accident of history that we have this name in much the same way that we still talk about "dialing" a number on our telephones even though modern telephones don't have a dial. The C programming language introduced a type called float for storing real numbers (short for "floating point number"). But floats had limited accuracy and another type was introduced called double, short for "double precision" (double the precision of a simple float). As memory has become cheaper, people have moved more towards using double as the default for floating-point values. In hindsight, it might have been better to use the word float for what is now called double and then used a word like "half" for the values with less accuracy, but it's tough to change habits that are so ingrained. So programming languages will continue to use the word double for floating point numbers and people will still talk about dialing people on the phone even if they've never touched a telephone dial.

Expressions

When writing programs we will often need to include values and calculations. The technical term for these is *expressions*.

Expression

A simple value or a set of operations that produces a value.

The simplest expression is a specific value, like 42 or 28.9. We call these "literal values" or *literals*. More complex expressions involve combining simple values. Suppose, for example, that you want to know how many bottles of water you have. If you have two 6-packs, four 4-packs and 2 individual bottles, then you could compute the total number of bottles with the following expression:

```
(2 * 6) + (4 * 4) + 2
```

Notice that we use the asterisk to represent multiplication and that we use parentheses to group parts of the expression. The computer determines the value of an expression by evaluating it.

Evaluation

The process of obtaining the value of an expression.

The value obtained when an expression is evaluated is called the *result*.

Complex expressions are formed using *operators*.

Operator

A special symbol (like `+` or `*`) used to indicate an operation to be performed on one or more values.

The values used in the expression are called *operands*. For example, consider the following simple expressions:

```
3 + 29
4 * 5
```

The operators here are the + and * and the operands are simple numbers:

```
   3        +        29
   |        |         |
operand  operator  operand

   4        *         5
   |        |         |
operand  operator  operand
```

When you form complex expressions, these simpler expressions can in turn become operands for other operators. For example, consider the following expression:

```
(3 + 29) - (4 * 5)
```

Which has two levels of operators:

```
  (3        +        29)     -      (4        *         5)
   |        |         |      |       |        |         |
operand  operator  operand   |    operand  operator  operand
   |                 |      |       |                  |
   +-----------------+      |       +------------------+
         operand          operator          operand
```

The plus operator has simple operands of 3 and 29 and the times operator has simple operands of 4 and 5, but the minus operator has operands that are each parenthesized expressions with operators of their own. Thus, complex expressions can be built from smaller expressions. At the lowest level you have simple numbers. These are used as operands to make more complex expressions, which in turn can be used as operands in even more complex expressions.

There are many things you can do with expressions. One of the simplest things you can do is to print the value of an expression using a println statement. For example, if you say:

```
System.out.println(42);
System.out.println(2 + 2);
```

you will get the following two lines of output:

```
42
4
```

Notice that for the second println, the computer evaluates the expression (adding 2 and 2) and prints the result (in this case, 4).

You will see many different operators as you progress through the book, all of which can be used to form expressions. Expressions can be arbitrarily complex, with as many operators as you like. For that reason, when we tell you, "An expression can be used here," we will often point out that we mean "arbitrary expressions" to emphasize that you can use complex expressions as well as simple values.

Literals

The simplest expressions refer to values directly using what are known as literals. An integer literal (considered to be of type int) is a sequence of digits with or without a leading sign:

```
3    482    -29434    0    92348    +9812
```

A floating point literal (considered to be of type double) will include a decimal point, as in:

```
298.4    0.284    207.    .2843    -17.452    -.98
```

Notice that `207.` is considered a double even though it coincides with an integer because of the decimal point. Literals of type double can also be expressed in scientific notation (a number followed by `e` followed by an integer) as in:

```
2.3e4    1e-5    3.84e92    2.458e12
```

The first number above represents 2.3 times 10 to the 4th power, which equals 23 thousand. Even though this happens to coincide with an integer, it is considered to be a value of type double because it is expressed in scientific notation. The second number represents 1 times 10 to the -5 power, which is equal to 0.00001. The third value represents 3.84 times 10 to the 92nd power. The fourth represents 2.458 times 10 to the 12th power.

Character literals (of type char) are enclosed in single quotation marks and can include just one character:

```
'a'    'm'    'X'    '!'    '3'    '\\'
```

All of these are of type char. Notice that the last example uses an escape sequence to represent the backslash character. You can even refer to the single quotation character using an escape sequence:

```
'\''
```

Remember that the primitive type boolean stores logical information. Logic deals with just two possibilities: true and false. These two words are keywords in Java that are the two literal values of type boolean:

```
true    false
```

Arithmetic Operators

The basic arithmetic operators are as follows.

Arithmetic Operators in Java

Operator	Meaning	Example	Result
+	addition	2 + 2	4
-	subtraction	53 - 18	35
*	multiplication	3 * 8	24
/	division	4.8 / 2.0	2.4
%	remainder or mod	19 % 5	4

The addition and subtraction operators should be familiar to you. The asterisk as a multiplication operator might be a surprise for nonprogrammers but doesn't take long to get used to. Division is also fairly familiar, although as we'll see, Java has two different division operations. The remainder or mod operation is the one that is most likely to be unfamiliar to you.

Division presents a problem when the operands are integers. When you divide 119 by 5, for example, you do not get an integer result. Therefore, integer division is expressed as two different integers--a quotient and a remainder:

```
119
---  =  23 (quotient) with 4 (remainder)
 5
```

In terms of the arithmetic operators:

```
119 / 5 evaluates to 23
119 % 5 evaluates to  4
```

Long division calculations are performed like this:

```
       31
   _______
34 ) 1079
     102
     ----
       59
       34
       --
       25
```

Here, dividing 1079 by 34 yields 31 with a remainder of 25. Using arithmetic operators, the problem would be described like this:

```
1079 / 34 evaluates to 31
1079 % 34 evaluates to 25
```

It takes a while to get used to integer division. For the division operator `/`, the key thing to keep in mind is that it truncates anything after the decimal point. So if you imagine computing an answer on a calculator, just think of ignoring anything after the decimal point.

```
19 / 5 is 3.8 on a calculator, so 19 / 5 evaluates to 3
207 / 10 is 20.7 on a calculator, so 203 / 10 evaluates to 20
3 / 8 is 0.375 on a calculator, so 3 / 8 evaluates to 0
```

The remainder operator is usually referred to as the "mod operator" or simply "mod". The mod operator lets you know how much was left unaccounted for by the truncated division operator. You have to think in terms of what the truncated division operator gives as a result and then you can determine what is left over. For example, given the examples above, we'd compute the mod results as follows.

```
  Mod           First       What does division   How much is
Problem         divide          account for?       left over?     Answer?
-------------------------------------------------------------------------
 19 % 5       19 / 5 is 3      3 * 5 is 15      19 - 15 is 4        4
207 % 10     207 / 10 is 20   20 * 10 is 200   207 - 200 is 7       7
 3 % 8         3 / 8 is 0       0 * 8 is 0       3 - 0 is 3         3
```

In each case, we figure out how much of the number is accounted for by the truncated division operator. The mod operator gives you any excess (the remainder). Putting the table above into a formula, you can think of the mod operator as behaving as follows:

```
x % y = x - (x / y) * y
```

You can get a result of 0 for the mod operator. This happens when one number goes evenly into another. For example, each of the following expressions evaluates to 0 because the second number goes evenly into the first number.

```
28 % 7
95 % 5
44 % 2
```

The mod operator has many useful applications in computer programs. Here are just a few ideas.

- testing whether a number is even or odd (number % 2 is 0 for evens, number % 2 is 1 for odds)
- finding individual digits of a number (e.g., number % 10 is the final digit)
- finding the last four digits of a social security number (number % 10000)

For floating point values (values of type double), the division operator does what we consider "normal" division. So even though the expression 119/5 evaluates to 23, the expression 119.0/5.0 evaluates to 23.8. The remainder operator can be used with doubles as well as with ints and it has a similar meaning. You consider how much is left over when you take away as many "whole" values as you can. For example, the expression 10.2 % 2.4 evaluates to 0.6 because you can take away four 2.4's from 10.2, leaving you with 0.6 left over.

Precedence

Java expressions are like complex noun phrases in English. Such phrases are subject to ambiguity, as in, "the man on the hill by the river with the telescope." Is the river by the hill or by the man? Is the man holding the telescope, or is the telescope on the hill, or is the telescope in the river? You don't know how to group the various parts together.

You can get the same kind of ambiguity if parentheses aren't used to group the parts of a Java expression. For example, the expression 2 + 3 * 4 has two operators. Which is performed first? You could interpret this two ways:

```
2 + 3 * 4         2 + 3 * 4
\---/                 \---/
  5   * 4         2 +  12
  \-----/         \-----/
    20              14
```

The first of these evaluates to 20 while the second evaluates to 14. To deal with the ambiguity, Java has rules of *precedence* that determine how to group the various parts together.

> **Precedence**
>
> The binding power of an operator which determines how to group parts of an expression.

Rules of precedence are applied when the grouping of operators in an expression is ambiguous. An operator with low precedence is evaluated after operators of higher precedence. Within a given level of precedence the operators are evaluated in one direction, usually left to right.

For arithmetic expressions there are two levels of precedence. The multiplicative operators (*, /, %) have a higher level of precedence than the additive operators (+, -). So the expression 2 + 3 * 4 is interpreted as:

```
2 + 3 * 4
    \---/
2 +  12
\-----/
  14
```

Within the same level of precedence, arithmetic operators are evaluated from left to right. This often doesn't make a difference in the final result, but occasionally it does. Consider, for example, the expression:

```
40 - 25 - 9
```

which evaluates as follows:

```
40 - 25 - 9
\-----/
  15    - 9
  \-------/
      6
```

You would get a different result if the second minus were evaluated first.

You can always override precedence with parentheses. For example, if you really want the second minus to be evaluated first, you can force that to happen by introducing parentheses:

```
40 - (25 - 9)
```

which evaluates as follows:

```
40 - (25 - 9)
     \------/
40 -    16
\--------/
    24
```

With arithmetic we also have what are known as "unary" plus and minus, with a single operand. For example, we can find the negation of 8 by asking for `-8`. These unary operators have a higher level of precedence, which allows you to form expressions like the following:

```
12 * -8
```

which evaluates to -96.

We will see many operators in the next few chapters. The following is a precedence table that has the arithmetic operators. As we learn about more operators, we'll update this table to include them as well. The table is ordered from highest precedence to lowest precedence. For example, the table indicates that Java will first group parts of an expression using the unary operators, then it will group using the multiplicative operators and only then will it group using the additive operators.

Java Operator Precedence

Description	Operators
unary operators	`+, -`
multiplicative operators	`*, /, %`
additive operators	`+, -`

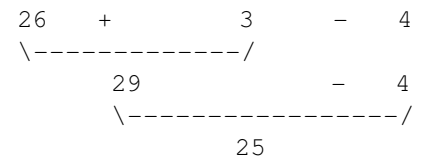
Before we leave this topic, let's look at a complex expression and see how it is evaluated step by step. Consider the following expression:

```
13 * 2 - 209 / 10 % 5 + 2 * 2
```

It has a total of six operators: two multiplications, one division, one mod, one subtraction and one addition. The multiplications, division and mod will be performed first because they have higher precedence and they will be performed from left-to-right order because they are all at the same level of precedence:

```
13 * 2 + 239 / 10 % 5 - 2 * 2
\----/
  26   + 239 / 10 % 5 - 2 * 2
         \------/
  26   +    23    % 5 - 2 * 2
            \-------/
  26   +         3     - 2 * 2
                         \---/
  26   +         3     -   4
```

Now we evaluate the additive operators from left to right:



Mixing Types and Casting

We often find ourselves mixing values of different types and wanting to convert from one type to another. Java has simple rules that avoid confusion and provides a mechanism for requesting that a value be converted from one type to another.

We often find ourselves mixing ints and doubles. We might, for example, ask Java to compute 2 * 3.6. This expression includes the int literal 2 and the double literal 3.6. In this case Java converts the int into a double and performs this computation entirely with double values. If Java encounters an int where it was expecting a double, it always converts the int to a double.

This becomes particularly important when you form expressions that involve division. If the two operands are both of type int, then Java will use integer (truncated) division. If either of the two operands is of type double, however, then it will do real-valued (normal) division. For example, 23 / 4 evaluates to 5 but all of the following evaluate to 5.75:

```
23.0 / 4
23. / 4
23 / 4.0
23 / 4.
23. / 4.
23.0 / 4.0
```

Sometimes you want Java to go the other way, converting a double into an int. You can ask Java for this conversion with a *cast*. Think of it as "casting a value in a different light." You request a cast by putting the name of the type you want to cast to in parentheses in front of the thing you want to cast. For example, if you say:

```
(int) 4.75
```

you will get the int value 4. When you cast a double value to an int it simply truncates anything after the decimal point. If you want to cast the result of an expression, you have to be careful to use parentheses. For example, suppose that you have some books that are 0.15 feet wide and you want to know how many of them would fit in a bookshelf that is 2.5 feet wide. You could do a straight division of 2.5 / 0.15, but that evaluates to a double result that is between 16 and 17. Americans use the phrase "16 and change" as a way to express the idea that it is larger than 16 but not as big as 17. In this case, we don't care about the "change." We want to compute the 16 part. You might form the following expression:

```
(int) 2.5 / 0.15
```

Unfortunately, this expression evaluates to the wrong answer because the cast is applied to whatever comes right after it, which is the value 2.5. So this casts 2.5 into the integer 2, divides by 0.15 and evaluates to 13 and change, which isn't an integer and isn't the right answer. You want to form this expression:

```
(int) (2.5 / 0.15)
```

This expression performs the division first to get 16 and change and then it casts that value to an int by truncating and evaluates to the int value 16, which is the answer you're looking for.

In later chapters we'll see other expressions involving mixed types. For example, in Chapter 4 we will see that every value of type char has a corresponding integer value, which allows for strange expressions like the following:

```
2 * 'a'                 // produces 194
(char) ('f' + 2)        // produces 'h'
```

2.2 Variables

Primitive data can be stored in the computer's memory in a *variable*.

> **Variable**
>
> A memory location with a name and a type that stores a value.

Think of the computer's memory as being like a giant spreadsheet that has many cells where data can be stored. When you tell Java that you want to have a variable, you are asking it to set aside one of those cells for this new variable. Initially the cell will be empty, but you will have the option to store a value in the cell. And as with spreadsheets, you will have the option to change the value in that cell later if you want to.

Java is a little more picky than a spreadsheet in that it requires you to tell it exactly what kind of data you are going to store in that cell. For example, if you want to store an integer, you need to tell Java that you intend to use type int. If you want to store a real value, you need to tell Java that you intend to use a double. You also have to decide on a name to use when you want to refer to this memory location. The normal rules of Java identifiers apply (must start with a letter and can then have any combination of letters and digits). The standard convention in Java is to start variable names with a lower case letter, as in `number` or `digits`, and to capitalize any subsequent words, as in `numberOfDigits`.

To explore the basic use of variables, let's examine a program that computes an individual's *body mass index* or BMI. Health professionals use this number to advise people about whether or not they are overweight. Given an individual's height and weight, we can compute a BMI. A simple BMI program, then, would naturally have three variables for these three pieces of information. There are several details that we need to discuss about variables, but it can be helpful to look at a complete program first to see the overall picture. The following is a complete program that computes and prints the BMI for an individual who is 5 foot 10 inches in height and weighs 195 pounds.

```
1  public class BMICalculator {
2      public static void main(String[] args) {
3          // declare variables
4          double height;
5          double weight;
6          double bmi;
7
8          // compute BMI
9          height = 70;
10         weight = 195;
11         bmi = weight / (height * height) * 703;
12
13         // print results
14         System.out.println("Current BMI:");
15         System.out.println(bmi);
16     }
17 }
```

Notice that the program includes blank lines and comments to indicate what the different parts of the program do. It produces the following output:

```
Current BMI:
27.976530612244897
```

Let's now examine the details of this program to understand how variables work. Before variables can be used in a Java program they must be declared. The line of code that declares the variable is known as a variable *declaration*.

Declaration

A request to set aside a new variable with a given name and type.

Each variable is declared just once. If you declare the variable more than once, you will get an error message from the Java compiler. Simple variable declarations are of the following form:

```
<type> <name>;
```

as in the three declarations at the beginning of our sample program:

```
double height;
double weight;
double bmi;
```

Notice that variable declarations end with a semicolon just like statements do. These declarations can appear anywhere a statement can occur. We are using the keyword double to define the type of these three variables. Remember that the name of each primitive type is a keyword in Java (int, double, char, boolean).

Notice that the declaration indicates the type and the name of the variable. Once a variable is declared, Java sets aside a memory location to store its value. Using this simple form of variable declaration, Java does not store an initial value in the memory location. So given the three declarations above, Java would allocate three memory locations for the variables and would have them initially not store a value:

```
       +---+               +---+            +---+
height| ? |        weight | ? |        bmi | ? |
       +---+               +---+            +---+
```

We refer to these as uninitialized variables and they are similar to blank cells in a spreadsheet. So how do we get values into those cells? The easiest way to do so is using an *assignment statement*. The general syntax of the assignment statement is:

```
<variable> = <expression>;
```

as in:

```
height = 70;
```

This statement stores the value 70 in the memory location for the variable height indicating that this person is 70 inches tall (5 foot 10 inches). We often use the phrase "gets" or "is assigned" when reading a statement like this, as in "height gets 70" or "height is assigned 70".

When the statement executes, the computer first evaluates the expression on the right side; then, it stores the result in the memory location for the given variable. In this case the expression is just a simple literal value 70, so after executing this statement, memory would look like this:

```
       +------+               +---+            +---+
height | 70.0 |        weight | ? |        bmi | ? |
       +------+               +---+            +---+
```

Notice that the value is stored as 70.0 because the variable is of type double. The variable height has now been initialized, but the variables weight and bmi are still uninitialized. The second assignment statement gives a value to weight:

```
weight = 195;
```

After executing this statement memory looks like this:

```
       +------+               +-------+            +---+
height | 70.0 |        weight | 195.0 |        bmi | ? |
       +------+               +-------+            +---+
```

The third assignment statement includes a formula (an expression to be evaluated):

```
bmi = weight / (height * height) * 703;
```

To calculate the value of this expression, the computer takes the weight and divides by the square of the height and then multiplies by the literal value 703. The result is stored in the variable bmi. So memory looks like this after the third assignment statement (not all of the digits of bmi are included here):

```
       +------+               +-------+            +--------+
height | 70.0 |        weight | 195.0 |        bmi | 27.976 |
       +------+               +-------+            +--------+
```

The last two lines of the program report the BMI result using println statements:

```
System.out.println("Current BMI:");
System.out.println(bmi);
```

Notice that we can include a variable in a println statement the same way that we included literal values and other expressions to be printed.

As its name implies, a variable can take on different values at different times. For example, consider the following variation of the BMI program that computes a new BMI assuming the person lost 15 pounds (going from 195 pounds to 180 pounds).

```
 1  public class BMICalculator2 {
 2      public static void main(String[] args) {
 3          // declare variables
 4          double height;
 5          double weight;
 6          double bmi;
 7
 8          // compute BMI
 9          height = 70;
10          weight = 195;
11          bmi = weight / (height * height) * 703;
12
13          // print results
14          System.out.println("Current BMI:");
15          System.out.println(bmi);
16
17          // recompute BMI
18          weight = 180;
19          bmi = weight / (height * height) * 703;
20
21          // report new results
22          System.out.println("Target BMI:");
23          System.out.println(bmi);
24      }
25  }
```

The program begins the same way setting the three variables to the following values and reporting this initial value for BMI:

```
       +------+              +-------+           +--------+
height | 70.0 |       weight | 195.0 |       bmi | 27.976 |
       +------+              +-------+           +--------+
```

But the new program then includes the following assignment statement:

```
weight = 180;
```

This changes the value of the variable called weight:

```
       +------+              +-------+           +--------+
height | 70.0 |       weight | 180.0 |       bmi | 27.976 |
       +------+              +-------+           +--------+
```

You might think that this would also change the value of the variable bmi. After all, earlier in the program we said that the following should be true:

```
bmi = weight / (height * height) * 703;
```

But even though Java uses the equals sign for assignment, don't confuse this with a statement of equality. The assignment statement does not represent an algebraic relationship. In algebra you might say:

```
x = y + 2
```

In mathematics you state definitively that x is equal to y plus 2, a fact that is true now and forever. If x changes, y will change accordingly. Java's assignment statement is very different.

The assignment statement is a command to perform an action at a particular point in time. It does not represent a lasting relationship between variables. That's why we usually use say "gets" or "is assigned" rather than saying "equals" when we read assignment statements.

Getting back to our program, resetting the variable called weight does not reset the variable called bmi. That's why the new program resets the weight *and* recomputes the bmi so that they each get a new value:

```
weight = 180;
bmi = weight / (height * height) * 703;
```

Without the second assignment statement above, the variable bmi would store the same value as before. That would be a rather depressing outcome to report to the person that losing 15 pounds will have no effect on their BMI. With the command to recompute, we reset both the weight and bmi variables so that memory looks like this (again, we are showing only the leading digits of bmi):

```
       +------+              +-------+            +--------+
height | 70.0 |        weight | 180.0 |        bmi | 25.824 |
       +------+              +-------+            +--------+
```

The output of the new version of the program is:

```
Current BMI:
27.976530612244897
Target BMI:
25.82448979591837
```

One very common assignment statement that points out the difference between algebraic relationships and program statements is:

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

Remember not to think of this as "x equals x + 1." There are no numbers that satisfy that equation. We use a word like "gets" to read this as, "x gets the value of x plus one." This may seem a rather odd statement, but you should be able to decipher it given the rules outlined above. Suppose that the current value of x is 19. To execute the statement, you first evaluate the expression to obtain the result 20. The computer stores this value in the variable named on the left, in variable x. Thus, this statement adds one to the value of the variable. We refer to this as *incrementing* the value of x. It is a fundamental programming operation because it is the programming equivalent of counting (1, 2, 3, 4 and so on). The following statement is a variation that counts down, which we call *decrementing* a variable:

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

Assignment/Declaration Variations

In the last section we saw the simplest form of variable declaration and assignment, but Java is a complex language that provides a lot of flexibility to programmers. It wouldn't be a bad idea to stick with the simplest form while you are learning, but you'll come across these other forms as you read other people's programs, so you'll want to understand what they mean.

The first variation is that Java allows you to provide an initial value to a variable at the time that you declare it. The syntax is as follows:

```
<type> <name> = <expression>;
```

as in:

```
double height = 70;
double weight = 195;
```

The statements above have the same effect as having two declarations followed by two assignment statements:

```
double height;
double weight;
height = 70;
weight = 195;
```

So this variation combines declaration and assignment in one line of code.

Another variation is to declare several variables all of the same type. The syntax is as follows:

```
<type> <name>, <name>, <name>, ..., <name>;
```

as in:

```
double height, weight;
```

Notice that the type appears just once at the beginning of the declaration. So the example above declares two different variables both of type double.

The final variation is a mixture of the previous two. You can declare multiple variables all of the same type and you can initialize them at the same time. For example, you could say:

```
double height = 70, weight = 195;
```

This not only declares the two double variables height and weight, it also gives them initial values (70 and 195, respectively). Java even allows you to mix initializing and not initializing, as in:

```
double height = 70, weight = 195, bmi;
```

This declares three double variables called height, weight and bmi and provides an initial value to two of them (height and weight). The variable bmi would be uninitialized by this declaration.

Common Programming Error: Accidentally Declaring a Variable Twice

One of the things to keep in mind as you learn is that you declare any given variable just once. You can assign it as many times as you like once you've declared it, but the declaration appears just once. Think of variable declaration as being like checking into a hotel and assignment as being like going in and out of your room. You have to check in first to get your room key, but then you can come and go as often as you like. If you tried to check in a second time, the hotel would be likely to ask you if you really want to pay for a second room.

If Java sees you declaring a variable more than once, it generates a compiler error. For example, if you say:

```
int x = 13;
System.out.println(x);
int x = 2;                // This line does not compile.
System.out.println(x);
```

The first line is okay. It declares an integer variable called x and initializes it to 13. The second line is okay because it simply prints the value of x. But the third line will generate an error message that "x is already defined". If you want to change the value of x, then you need to use a simple assignment statement instead of a variable declaration:

```
int x = 13;
System.out.println(x);
x = 2;
System.out.println(x);
```

We have been referring to the "assignment statement" but in fact assignment is an operator, not a statement. When you assign a value to a variable, the overall expression evaluates to the value just assigned. That means that you can form expressions that have assignment operators embedded within them. Unlike most other operators, the assignment operator evaluates from right to left, which allows programmers to write statements like the following:

```
int x, y, z;
x = y = z = 2 * 5 + 4;
```

Because the assignment operator evaluates right-to-left, this statement is equivalent to:

```
x = (y = (z = 2 * 5 + 4));
```

The expression `2 * 5 + 4` evaluates to 14. This value is assigned to z. But that in turn evaluates to 14, which is then used to assign a value to y. And that assignment evaluates to 14 as well, which is used to assign a value to x. So all three variables will be assigned the value 14 as a result of this statement.

While you can do assignments like these, it's not clear that it is wise to do so. The chained assignment statement above is not that difficult to read, but try to figure out what the following statement does:

```
x = 3 * (y = 2 + 2) / (z = 2);
```

It's easier to see what is going on when you write the code above as three separate statements:

```
y = 2 + 2;
z = 2;
x = 3 * y / z;
```

String Concatenation

You saw in chapter 1 that you can output string literals using System.out.println. You can also output numeric expressions using System.out.println:

```
System.out.println(12 + 3 - 1);
```

This statement causes the computer to first evaluate the expression, which yields the value 14, and then to write 14 to the console window. Often you want to output more than one value on a line. Unfortunately, you can only pass one value to println. To get around this limitation, Java provides a simple mechanism called *concatenation* for putting several pieces together into one long string literal.

String Concatenation

Combining several strings into a single string or combining a string with other data into a new longer string.

The plus operator concatenates the pieces together. Doing so forms an expression that can be evaluated. Even though such expressions include both numbers and text, they can be evaluated just like the numeric expressions we are exploring. Consider, for example, the following:

```
"I have " + 3 + " things to concatenate"
```

You have to pay close attention to the quotation marks in an expression like this to keep track of which parts are "inside" a string literal and which are outside. This expression begins with the text "I have " (including a space at the end). Then we see a plus sign and the integer literal 3. Java converts the integer into a textual form ("3") and concatenates the two pieces together to form "I have 3". Following the 3 is another plus and another string literal " things to concatenate" (which starts with a space). This piece is glued onto the end of the other string to form the string "I have 3 things to concatenate".

Because this expression produces a single concatenated string, we can include it in a println statement:

```
System.out.println("I have " + 3 + " things to concatenate");
```

which would produce a single line of output:

```
I have 3 things to concatenate
```

We often use string concatenation to report the value of a variable. Consider, for example, the following program that computes the number of hours, minutes and seconds in a standard year.

```
 1  public class Time {
 2      public static void main(String[] args) {
 3          int hours = 365 * 24;
 4          int minutes = hours * 60;
 5          int seconds = minutes * 60;
 6          System.out.println("Total hours in a year   = " + hours);
 7          System.out.println("Total minutes in a year = " + minutes);
 8          System.out.println("Total seconds in a year = " + seconds);
 9      }
10  }
```

Notice that the three println commands at the end each have a string literal concatenated with a variable. The program produces the following output.

```
Total hours in a year   = 8760
Total minutes in a year = 525600
Total seconds in a year = 31536000
```

You can use concatenation to form arbitrarily complex expressions. For example, if you had variables x, y and z and you wanted to write their values out in coordinate format with parentheses and commas you could say:

```
System.out.println("(" + x + ", " + y + ", " + z + ")");
```

If x, y and z have the values 8, 19 and 23, respectively, then this statement would output the string "(8, 19, 23)".

The plus used for concatenation has the same level of precedence as the normal arithmetic plus operator which can lead to some confusion. Consider, for example, the following expression:

```
2 + 3 + " hello " + 7 + 2 * 3
```

This expression has four plus operators and one multiplication operator. Because of precedence, we evaluate the multiplication first:

```
2 + 3 + " hello " + 7 + 2 * 3
                        \---/
2 + 3 + " hello " + 7 +   6
```

This grouping might seem odd, but that's what the precedence rule says to do. We don't evaluate any addition operators until we've first evaluated all of the multiplicative operators. So once we've taken care of the multiplication, we're left with the four addition operators. These will be evaluated from left to right.

In evaluating the first addition, we find ourselves asked to add together two integer values. The overall expression involves a string, but this little subexpression has just two integers. As a result, we perform integer addition.

```
2 + 3 + " hello " + 7 +   6
\---/
  5   + " hello " + 7 +   6
```

The next addition involves adding the integer 5 to the string literal " hello ". If either of the two operands is a string, then we perform concatenation. So in this case, we convert the integer into a text equivalent ("5") and glue the pieces together to form a new string value:

```
5   + " hello " + 7 +   6
\-------------/
  "5 hello "    + 7 +   6
```

You might think that Java would add together the 7 and 6 the way we did with the 2 and 3 that were added together to make 5. But it doesn't work that way. The rules of precedence are simple and Java follows them with simple-minded consistency. Precedence tells us that addition operators are evaluated left to right. So first we add the string "5 hello " to 7. That is another combination of a string and an integer, so Java converts the integer to its textual equivalent ("7") and concatenates the two parts together to form a new string:

```
"5 hello "    + 7 +   6
\---------------/
   "5 hello 7"    +   6
```

Now there is just a single addition to perform which again involves a string/int combination. We convert the integer to its textual equivalent ("6") and concatenate the two parts together to form a new string:

```
"5 hello 7"    +   6
\------------------/
    "5 hello 76"
```

Clearly such expressions can be confusing, but you wouldn't want the Java compiler to guess what you mean. Our job as programmers is easier if we know that the compiler is going to follow simple rules consistently. You can avoid much of this confusion by adding parentheses to the original expression. For example, if we really did want Java to add together the 7 and 6 instead of concatenating them separately, we could have written the original expression in a much clearer way as:

```
(2 + 3) + " hello " + (7 + 2 * 3)
```

Because of the parentheses, Java will evaluate the two numeric parts of this expression first and then concatenate the results with the string in the middle. This expression evaluates to "5 hello 13".

Increment/Decrement Operators

In addition to the standard assignment operator, Java has several special operators that are useful for a particular family of operations that are common in programming. You often find yourself increasing a variable by a particular amount which we call incrementing. You also often find yourself decreasing a variable by a particular amount, which we call decrementing. To accomplish this you write statements like the following:

```
x = x + 1;
y = y - 1;
z = z + 2;
```

You also often find yourself wanting to double or triple a variable or to reduce its value by a factor of 2, in which case you might write code like the following:

```
x = x * 2;
y = y * 3;
z = z / 2;
```

Java has a shorthand for these situations. You glue together the operator character (+, -, *, etc) with the equals to get a special assignment operator (+=, -=, *=, etc). This variation allows you to rewrite assignments statements like the ones above as follows:

```
x += 1;
y -= 1;
z += 2;

x *= 2;
y *= 3;
z /= 2;
```

This convention is yet another detail to learn about Java, but the code can be clearer to read. Think of a statement like `x += 2` as saying "add 2 to x." That's more concise than saying `x = x + 2`.

Java has an even more concise way of expressing this for the particular case where you want to increment by 1 or decrement by 1 in which case you can use the increment and decrement operators (`++` and `--`). For example, you can say:

```
x++;
y--;
```

There are actually two different forms of each of these because you are also allowed to put the operator in front of the variable:

```
++x;
--y;
```

The two versions of `++` are known as the pre-increment (++x) and post-increment (x++) operators. The two versions of `--` are similarly known as the pre-decrement (--x) and post-decrement (x--) operators. The pre versus post distinction doesn't matter when you include them as statements by themselves, as in these two examples. The difference comes up only when you embed these inside of more complex expressions, which we don't recommend.

Now that we've seen a number of new operators, it is worth revisiting the issue of precedence. Here is an updated precedence table including the assignment operators and the increment and decrement operators.

Java Operator Precedence

Description	Operators
unary operators	`++, --, +, -`
multiplicative operators	`*, /, %`
additive operators	`+, -`
assignment operators	`=, +=, -=, *=, /=, %=`

Did you Know: ++ and --

The `++` and `--` operators were first introduced in the C programming language. Java has them because the designers of the language decided to use the syntax of C as the basis for Java syntax. Many languages have made the same choice, including C++ and C#. There is almost a sense of pride among C programmers that these operators allow you to write extremely concise code. Many other people feel that they can make code unnecessarily complex. In this book we always use these operators as a separate statement so that it is obvious what is going on, but some people like to know all of the details, so here they are.

The pre and post variations both have the same overall effect. The two increment operators increment a variable. The two decrement operators decrement a variable. But they differ in terms of what they evaluate to. When you increment or decrement, there are really two values involved. There is the original value that the variable had before the increment or decrement and there is the final value that the variable has after the increment or decrement. The post versions evaluate to the original (older) value and the pre versions evaluate to the final (later) value.

Consider, for example, the following code fragment:

```
int x = 10;
int y = 20;
int z = ++x * y--;
```

What value is z assigned? The answer is 220. The third assignment increments x to 11 and decrements y to 19, but in computing the value of z, it uses the new value of x (++x) times the old value of y (y--), which is 11 times 20 or 220.

There is a simple mnemonic to remember this. When you see `x++` read it as "give me x, then increment" and read `++x` as "increment, then give me x." Here's another memory device that might help. Just remember that C++ is a bad name for a programming language. The expression "C++" would be interpreted as, "Evaluate to the old value of C and then increment C." In other words, even though you're trying to come up with something new and different, you're really stuck with the old awful language. The language you want is ++C, because then you'd improve the language and you'd get to work with the new and improved language rather than the old one. Some people have suggested that perhaps Java is ++C.

Variables and Mixing Types

We have seen that when you declare a variable, you must tell Java what type of value it will be storing. For example, you might declare a variable of type int for integer values or of type double for real values. The situation is fairly clear when you have just integers or just reals, but what happens when you start mixing the types? For example, the following code is clearly okay.

```
int x;
double y;
x = 2 + 3;
y = 3.4 * 2.9;
```

We have an integer variable that we assign an integer value and a double variable that we assign a double value. But what if we try to do it the other way around?

```
int x;
double y;
x = 3.4 * 2.9;  // illegal
y = 2 + 3;      // okay
```

As the comments indicate, you can't assign an integer variable a double value, but you can assign a double variable an integer value. Let's consider the second case first. The expression `2 + 3` evaluates to the integer 5. This value isn't a double, but every integer is a real value, so it is easy enough for Java to convert the integer into a double. The technical term is that Java "promotes" the integer into a double.

The other direction is more problematic. The expression `3.4 * 2.9` evaluates to the double value 9.86. This value can't be stored in an integer because it isn't an integer. If you wanted to perform this kind of operation, you'd have to tell Java how to convert this into an integer. As described earlier, you can cast to an int, which will truncate anything after the decimal point:

```
x = (int) (3.4 * 2.9);  // now legal
```

This statement first evaluates 3.4 * 2.9 to get 9.86 and then truncates to get the integer 9.

Common Programming Error: Forgetting to Cast

We often write programs that involve a mixture of ints and doubles, so it is easy to make mistakes when it comes to combinations of the two. For example, suppose that we want to compute a student's percent right on a test given the total number of questions on the test and the number of questions the student got right. We might declare the following variables:

```
int totalQuestions;
int numRight;
double percent;
```

Suppose the first two are initialized as follows:

```
totalQuestions = 73;
numRight = 59;
```

How do we compute the percent that the student got right? We divide the number right by the total number of questions and multiply by 100 to turn it into a percentage:

```
percent = numRight / totalQuestions * 100;
```

Unfortunately, if we print out the value of the variable percent after executing this line of code, we will find that it has the value 0.0. Obviously the student got more than 0% correct.

The problem comes from integer division. The expression we are using begins with two int values:

```
numRight / totalQuestions
```

Which means we are computing:

```
59 / 73
```

This evaluates to 0 with integer division. Some students fix this by changing the types of the variables to be all doubles. That will solve the immediate problem, but it's not a good choice to make from a stylistic point of view. It is best to use the most appropriate type for data and we certainly expect that the number of questions on the test will be an integer.

We could try to fix this by changing the value 100 to 100.0:

```
percent = numRight / totalQuestions * 100.0;
```

This doesn't help because we do the division first. One version that does work is to put the 100.0 first:

```
percent = 100.0 * numRight / totalQuestions;
```

This works because now the multiplication is computed before the division, which means that everything is converted to double. Sometimes you can fix a problem like this through a clever rearrangement of the formula, but you don't want to count on cleverness. This is a good place to use a cast. For example, returning to our original formula, we can cast each of the int variables into double:

```
percent = (double) numRight / (double) totalQuestions * 100.0;
```

We can also take advantage of the fact that once we have cast one of these two variables to double, the division will be done with doubles. So we could, for example, cast just the first value to double:

```
percent = (double) numRight / totalQuestions * 100.0;
```

2.3 The for Loop

Programming often involves specifying redundant tasks. The for loop helps to avoid such redundancy. Suppose you want to write out the squares of the first 5 integers. You can say:

```
1  public class WriteSquares {
2      public static void main(String[] args) {
3          System.out.println(1 + " squared = " + (1 * 1));
4          System.out.println(2 + " squared = " + (2 * 2));
5          System.out.println(3 + " squared = " + (3 * 3));
6          System.out.println(4 + " squared = " + (4 * 4));
7          System.out.println(5 + " squared = " + (5 * 5));
8      }
9  }
```

This program produces the following output:

```
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
```

This approach is tedious. The program has five statements that are very similar. They are all of the form:

```
System.out.println(number + " squared = " + (number * number));
```

where number is either 1, 2, 3, 4, or 5. The for loop avoids such redundancy. Here is an equivalent program using a for loop:

```
1  public class WriteSquares2 {
2      public static void main(String[] args) {
3          for (int i = 1; i <= 5; i++) {
4              System.out.println(i + " squared = " + (i * i));
5          }
6      }
7  }
```

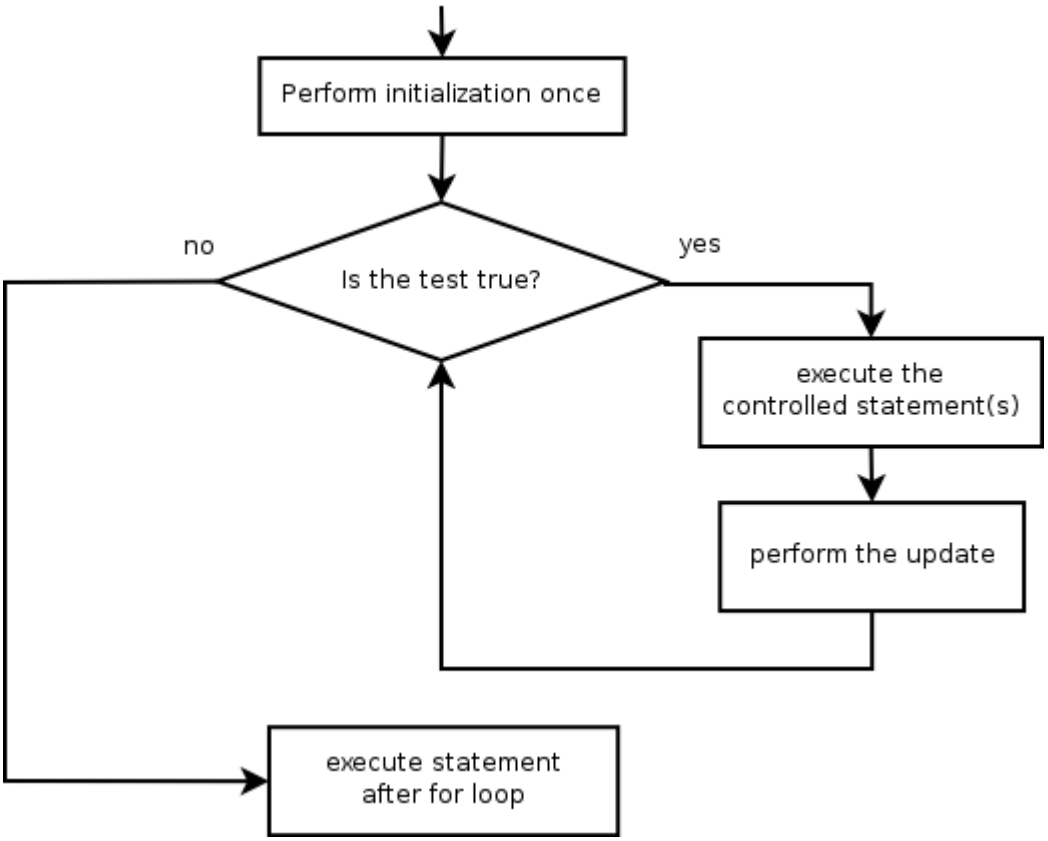
This program initializes a variable called i to the value 1. Then it repeatedly executes the println statement as long as the variable i is less than or equal to 5. After each println, it evaluates the expression i++ to increment i.

The general syntax of the for loop is as follows:

```
for (<initialization>; <continuation test>; <update>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

You always include the keyword for and the parentheses. Inside the parentheses you have three different parts separated by semicolons: the initialization, the continuation test and the update. Then you have a set of curly braces that enclose a set of statements. The for loop controls these statements inside the curly braces. We refer to the controlled statements as the *body* of the loop. The idea is that we execute the body multiple times, as determined by the combination of the other three parts.

The following diagram indicates the steps that Java follows to execute a for loop. It performs whatever initialization you have requested once before the loop begins executing. Then it repeatedly performs the continuation test you have provided. If the continuation test evaluates to true, then it executes the controlled statements once and executes the update part. Then it performs the test again and if it again evaluates to true, it executes the statements again and evaluates the update again. Notice that the update is performed after the controlled statements are executed. When the test evaluates to false, Java is done executing the loop and moves on to whatever statement comes after the loop.



The for loop is the first example of a *control structure*.

Control Structure

A syntactic structure that controls other statements.

You should be careful to use indentation to indicate controlled statements. In the case of the for loop, we indent all of the statements in the body of the loop as a way to indicate that they are "inside" the loop.

Tracing for Loops

Let's examine the for loop of the WriteSquares2 program in detail:

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

In this loop the initialization (int i = 1) declares an integer variable i that is initialized to 1. The continuation test (i <= 5) indicates that we should keep executing as long as i is less than or equal to 5. That means that once i is greater than 5, we will stop executing the body of the loop. The update (i++) will increase i by one each time, getting i closer to being larger than 5. After five executions of the body and the accompanying five updates, i will be larger than 5 and the loop will finish executing. The following is a detailed trace of this process.

```
Step              Code          Description
-------------------------------------------------------------------------
initialization    int i = 1;    variable i is allocated and initialized to 1

test              i <= 5        true because 1 <= 5, so we enter the loop
body              {...}         execute the println with i equal to 1
update            i++           increment i, which becomes 2

test              i <= 5        true because 2 <= 5, so we enter the loop
body              {...}         execute the println with i equal to 2
update            i++           increment i, which becomes 3

test              i <= 5        true because 3 <= 5, so we enter the loop
body              {...}         execute the println with i equal to 3
update            i++           increment i, which becomes 4

test              i <= 5        true because 4 <= 5, so we enter the loop
body              {...}         execute the println with i equal to 4
update            i++           increment i, which becomes 5

test              i <= 5        true because 5 <= 5, so we enter the loop
body              {...}         execute the println with i equal to 5
update            i++           increment i, which becomes 6

test              i <= 5        false because 6 > 5, so we are finished
```

Notice from this trace that the body of the loop (the println) is executed a total of 5 times.

Java allows great flexibility in deciding what to include in the initialization part and the update which allows us to use the for loop to solve all sorts of programming tasks. For right now, we will restrict ourselves to a particular kind of loop that declares and initializes a single variable that is used to control the loop. This variable is often referred to as the control variable of the loop. In the test we compare the control variable against some final desired value and in the update we change the value of the control variable, most often incrementing it by 1. Such loops are very common in programming. By convention, we often use variable names like i, j and k for such loops.

Each execution of the controlled statement of a loop is called an iteration of the loop, as in, "The loop finished executing after four iterations." Iteration also refers to looping in general, as in, "I solved the problem using iteration."

Consider another for loop:

```
for (int i = -100; i <= 100; i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

This loop executes a total of 201 times producing the squares of all the integers between -100 and +100 inclusive. The values used in the initialization and the test, then, can be any integers. They can, in fact, be arbitrary integer expressions:

```
for (int i = (2 + 2); i <= (17 * 3); i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

This loop will generate the squares between 4 and 51 inclusive. The parentheses around the expressions are not necessary, but improve readability. Consider the following loop:

```
for (int i = 1; i <= 30; i++) {
    System.out.println("+--------+");
}
```

This loop generates 30 lines of output, all exactly the same. This loop is slightly different from the previous one because the statement controlled by the for loop makes no reference to the control variable. Thus:

```
for (int i = -30; i <= -1; i++) {
    System.out.println("+--------+");
}
```

generates exactly the same output. The behavior of such a loop is determined solely by the number of iterations it performs. The number of iterations is given by:

```
(ending value) - (starting value) + 1
```

It is much simpler to see that the first of these loops iterates 30 times, so it is better to use. In general, if we want a loop to iterate exactly n times, we will use one of two standard loops. The first standard form looks like the ones we have seen above:

```
for (int <variable> = 1; <variable> <= n; i++) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

It's pretty clear that this loop executes n times because it starts at 1 and continues as long as it is less than or equal to n. Often, however, it is more convenient to start our counting at 0 instead of 1. That requires a change in the loop test to allow us to stop when n is 1 less:

```
for (int <variable> = 0; <variable> < n; i++) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

Notice that in this form when we initialize the variable to 0, we test whether it is strictly less than n. Either form will execute exactly n times, although we will see some situations where the 0-based loop works better.

Let's consider some borderline cases. What happens if you say:

```
for (int i = 1; i <= 1; i++) {
    System.out.println("+--------+");
}
```

According to our rule, it should iterate once and it does. It initializes the variable i to 1 and tests to see if this is less than or equal to 1, which it is. So it executes the println, increments i and tests again. The second time it tests, it finds that i is no longer less than or equal to 1, so it stops executing. What about this loop:

```
for (int i = 1; i <= 0; i++) {
    System.out.println("+--------+");  // never executes
}
```

This loop performs no iterations at all. It will not cause an execution error; it just won't execute the body. The variable is initialized to 1 and we test to see if it is less than or equal to 0. It isn't, so we don't execute the statements in the body at all.

When you construct a for loop, you can include more than one statement inside the curly braces. Consider, for example, the following code:

```
for (int i = 1; i <= 20; i++) {
    System.out.println("Hi!");
    System.out.println("Ho!");
}
```

This will produce 20 pairs of lines, the first of which has the word "Hi" on it and the second of which has the word "Ho".

When a for loop controls a single statement, you don't have to include the curly braces. The curly braces are required only for situations like the one above where you have more than one statement that you want the loop to control. The Sun coding convention includes the curly braces even for a single statement and we follow this convention in this book. There are several advantages to this convention:

- Including the curly braces prevents future errors. Even if you need only one statement in the body of your loop now, your code is likely to change over time. Having the curly braces there makes it less likely that you will accidentally add an extra statement to the body later and forget to include curly braces. In general, including curly braces in advance is cheaper than locating obscure bugs later.
- Always including the curly braces reduces the level of detail you have to consider as you learn new control structures. It takes time to master the details of any new control structure. It will be easier to master those details if you don't have to also be thinking about when to include and when not to include the braces.

Common Programming Error: Forgetting Curly Braces

You should use indentation to indicate the body of a for loop, but indentation alone is not enough. Java ignores indentation when deciding how different statements are grouped. Suppose, for example, that you were to write the following code:

```
for (int i = 1; i <= 20; i++)
    System.out.println("Hi!");
    System.out.println("Ho!");
```

The indentation indicates that both of the println statements are in the body of the for loop. But there aren't any curly braces to indicate that to Java. As a result, this code is interpreted as follows:

```
for (int i = 1; i <= 20; i++) {
    System.out.println("Hi!");
}
System.out.println("Ho!");
```

Only the first println is considered to be in the body of the for loop. The second println is considered to be outside the loop. So this code would produce 20 lines of output that all say "Hi!" followed by one line of output that says "Ho!". To include both printlns in the body, we need curly braces around them:

```
for (int i = 1; i <= 20; i++) {
    System.out.println("Hi!");
    System.out.println("Ho!");
}
```

print Versus println

So far we have been producing entire lines of output using println commands. Now that we now how to write for loops, we will want to be able to produce complex lines of output piece by piece. For example, if we want to produce a line of output that has 80 stars on it, it would be easier to use a loop that prints one star at a time and have it execute 80 times rather than trying to use a single println. Before we can do that, we have to learn about a variation of the println command.

Java has a variation of the println command called print that allows you to produce output on the current line without going to a new line of output. The println command really does two different things: it sends output to the current line and then it moves to the beginning of a new line. The print command does only the first of these. Thus, a series of print commands will generate output all on the same line. Only a println command will cause the current line to be completed and a new line to be started. For example, consider these six statements:

```
System.out.print("Hi Ho, ");
System.out.print("my man.");
System.out.print("Where are you bound?  ");
System.out.println("My way, I hope.");
System.out.print("This is");
System.out.println(" for the whole family!");
```

These statements produce two lines of output. Remember every println statement produces exactly one line of output. Because there are two println statements here, there are two lines of output. After the first statement executes, the current line looks like this:

```
Hi ho,
       ^
```

The arrow below the output line indicates the position where output will be sent next. We can simplify our discusion if we refer to the arrow as the *output cursor*. Notice that the output cursor is at the end of this line and that it is preceded by a space. That is so because the command was a print (don't go to a new line) and the string literal in the print ends with a space. Java will not insert a space for you unless you specifically request it. After the next print, the line looks like this:

```
Hi ho, my man.
              ^
```

The output cursor doesn't have a space before it now because the string literal in the print command ends in a period, not a space. After the next print, the line looks like this:

```
Hi ho, my man.Where are you bound?
                                  ^
```

There is no space between the period and the word "Where" because there was no space in the print commands. But the string literal in the third statement has spaces at the end and as a result the output cursor is positioned two spaces after the question mark. After the next statement executes, the output looks like this:

```
Hi ho, my man.Where are you bound?  My way, I hope.

^
```

Because this fourth statement is a println command, it finishes the output line and positions the cursor at the beginning of the second line. The next statement is another print that produces this:

```
Hi ho, my man.Where are you bound?  My way, I hope.
This is
       ^
```

The final println completes the second line and positions the output cursor at the beginning of a new line:

```
Hi ho, my man.Where are you bound?  My way, I hope.
This is for the whole family!

^
```

These six statements are equivalent to these two single statements:

```
System.out.println("Hi ho, my man.Where are you bound?  My way, I hope.");
System.out.println("This is for the whole family!");
```

It seems a bit silly to have both the print and the println commands for producing lines like these, but you will see that there are more interesting applications of print in combination with for loops.

Remember that it is possible to have an empty println command:

```
System.out.println();
```

Because there is nothing inside of parentheses to be written to the output line, this positions the output cursor to the beginning of the next line. If there are print commands before this empty println, it finishes out the line made by those print commands. If there are no previous print commands, it produces a blank line. An empty print command is meaningless and is illegal.

Nested for Loops

The for loop controls a statement, and the for loop is itself a statement, which means that one for loop can control another for loop. For example, we can write code like the following:

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.println("Hi there.");
    }
}
```

This code is probably easier to read from the inside out. The println statement produces a single line of output. The inner j loop executes this statement 5 times, producing 5 lines of output. The outer i loop executes the inner loop 10 times, which produces 10 sets of 5 lines, or 50 lines of output. The code above, then, is equivalent to:

```
for (int i = 1; i <= 50; i++) {
    System.out.println("Hi there.");
}
```

This example shows that a for loop can be controlled by another for loop. Such a loop is called a *nested loop*. This example wasn't very interesting because the nested loop can be eliminated. Let's look at a more interesting nested loop that does something useful:

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

When you write code that involves nested loops, you have to be careful to indent the code just right to make the structure clear. At the outermost level, the code above is a simple for loop that executes 6 times:

```
for (int i = 1; i <= 6; i++) {
    ...
}
```

We use indentation for the statements inside this for loop to make it clear that they are the body of this loop. Inside we find two statements: another for loop and a println. Let's look at the inner for loop:

```
    for (int j = 1; j <= 10; j++) {
        System.out.print("*");
    }
```

This loop is controlled by the outer for loop, which is why it is indented, but it itself controls a statement (the print statement), so we end up with another level of indentation to indicate that the print is controlled by the inner for loop, which in turn is controlled by the outer for loop. So what does this inner loop do? It prints ten stars on the current line of output. They all appear on the same line of output because we are using a print instead of a println. Notice that after this loop we perform a println:

```
System.out.println();
```

The net effect of the for loop followed by the println is that we get a line of output with 10 stars on it. But remember that these statements are contained in an outer loop that executes 6 times. So we end up getting 6 lines of output, each with 10 stars:

```
**********
**********
**********
**********
**********
**********
```

Let's examine one more variation. In the code above, the inner for loop always does exactly the same thing. It prints exactly 10 stars on a line of output. But what happens if we change the test for the inner for loop to make use of the outer for loop's control variable (i)?

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

In the old version the inner loop always executes 10 times producing 10 stars on each line of output. With the new test (j <= i), the inner loop is going to execute *i* times. But i is changing. It takes on the values 1, 2, 3, 4, 5 and 6. So on the first iteration of the outer loop when i is 1, the test j <= i is effectely testing j <= 1 and we produce a line with 1 star on it. On the second iteration of the outer loop when i is 2, the test is effectively testing j <= 2 and we produce a line with 2 stars on it. On the third iteration of the outer loop when i is 3, the test is effectively testing j <= 3 and we produce a line with 3 stars on it. And so on.

In other words, this code produces a triangle as output:

```
*
**
***
****
*****
******
```

2.4 Managing Complexity

Now that we have seen several new programming constructs, we are ready to put the pieces together to solve some complex tasks. As we pointed out in chapter 1, Brian Kernighan, one of the creators of the C programming language, has said that, "Controlling complexity is the essence of computer programming." In this section we will examine several techniques that computer scientists use to solve complex problems without being overwhelmed by complexity.

Scope

As programs get longer there is an increasing possibility of different part of the program interfering with each other. Java helps us to manage this potential problem by enforcing rules of *scope*.

Scope (of a declaration)

The part of a program in which a particular declaration is valid.

We have seen that when it comes to declaring static methods, we can put them in any order whatsoever. The scope of a static method is the entire class in which it appears. Variables work differently. The simple rule is that the scope of a variable declaration extends from the point where it is declared to the right curly brace that encloses it. In other words, find the pair of curly braces that directly enclose the variable declaration. The scope of the variable is from the point where it is declared to the closing curly brace.

There are several implications of this scope rule. Consider first what it means for different methods. Each method has its own set of curly braces to indicate the statements to be executed when the method is called. If variables are declared inside of a method's curly braces, then those variables won't be available outside the method. We refer to such variables as *local variables* and we refer to this process as *localizing* variables. In general, we want to declare variables in the most local scope possible.

Local Variable

A variable declared inside a method that is accessible only in that method.

Localizing Variables

Declaring variables in the innermost (most local) scope possible.

You might wonder why we would want to localize variables to just one method. Why not just declare everything in one outer scope? That certainly seems simpler. The idea is similar to the use of refrigerators in dormitories. Every dorm room can have its own refrigerator for use in that room. If you are outside of a room, you don't even know that it has a refrigerator in it. The contents of the room are hidden from you.

Localizing variables leads to some duplication and confusion, but provides more security. Our programs use variables to store values just as students use refrigerators to store beer, ice cream, and other valuables. The last time I was in a dorm I noticed that most of the individual rooms had refrigerators in them. This seems terribly redundant, but the reason is obvious. If you want to guarantee the security of something, you put it where nobody else can get it. You will use local variables in much the same way. Each individual method will have its own local variables to use, which means you don't have to consider possible interference from other parts of the program.

Let's look at a simple example involving two methods.

```
 1  // This program does not compile.
 2  public class ScopeExample {
 3      public static void main(String[] args) {
 4          int x = 3;
 5          int y = 7;
 6          computeSum();
 7      }
 8
 9      public static void computeSum() {
10          int sum = x + y;  // illegal because x and y are not in this scope
11          System.out.println("sum = " + sum);
12      }
13  }
```

In this example, the main method declares local variables x and y and gives them initial values. Then it calls method computeSum. Inside of method computeSum we try to make use of the values of x and y to compute a sum. But this doesn't work. The variables x and y are local to the main method and aren't visible inside of the computeSum method. In the next chapter we will see a technique for allowing one method to pass a value to another.

The program produces error messages like the following:

```
ScopeExample.java:10: cannot find symbol
symbol  : variable x
location: class ScopeExample
        int sum = x + y;  // illegal because x and y are not in this scope
                  ^
ScopeExample.java:10: cannot find symbol
symbol  : variable y
location: class ScopeExample
        int sum = x + y;  // illegal because x and y are not in this scope
                      ^
```

Scope rules are important to understand when we talk about the local variables of one method versus another method, but they also have implications for what happens inside a single method. We have seen that curly braces are used to group together a series of statements. But we can have curly braces inside of curly braces and this leads to some scope issues. For example, consider the following code:

```
for (int i = 1; i <= 5; i++) {
    int squared = i * i;
    System.out.println(i + " squared = " + squared);
}
```

This is a variation of the code we looked at earlier in the chapter to print out the squares of the first 5 integers. In this version, we use a variable called `squared` to keep track of the square of the for loop control variable. This code works fine, but consider this variation:

```
for (int i = 1; i <= 5; i++) {
    int squared = i * i;
    System.out.println(i + " squared = " + squared);
}
System.out.println("Last square = " + squared);  // illegal
```

This code generates a compiler error. The variable called squared is declared inside the for loop. In other words, the curly braces that contain it are the curly braces for the loop. It can't be used outside this scope. So when we attempt to refer to it outside the loop, we get a compiler error.

If for some reason you need to write code like this that accesses the variable after the loop, you'd have to declare the variable in the outer scope before the loop:

```
int squared;  // declaration is now in outer scope
for (int i = 1; i <= 5; i++) {
    squared = i * i;  // change this to an assignment statement
    System.out.println(i + " squared = " + squared);
}
System.out.println("Last square = " + squared);  // now legal
```

There are a few special cases for scope and the for loop is one of them. When a variable is declared in the initialization part of a for loop, then its scope is just the for loop itself (the 3 parts in the for loop header and the statements controlled by the for loop). That means that you can use the same variable name in multiple for loops:

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i + " squared = " + (i * i));
}
for (int i = 1; i <= 10; i++) {
    System.out.println(i + " cubed = " + (i * i * i));
}
```

The variable i is declared twice, but because the scope of each is just the for loop in which it is declared, this isn't a problem. This is like having two dorm rooms, each with its own refrigerator. Of course, we can't do this with nested for loops:

```
for (int i = 1; i <= 5; i++) {
    for (int i = 1; i <= 10; i++) {
        System.out.println("hi there.");
    }
}
```

This code won't compile. When Java encounters the inner for loop, it will complain that the variable i has already been declared within this scope. You aren't allowed to declare the same variable twice within the same scope. You'd have to come up with two different names. This practice is similar to what families do when they have two people with the same name. The families change the names to be able to distinguish between the two people. So you end up with "Carl Junior" and "Carl Senior" or "Grandma Jones" and "Grandma Smith" to avoid any potential confusion.

In all of the for loop examples we have looked at, the control variable is declared in the initialization part of the loop. This isn't a requirement. You can separate the declaration of the for loop control variable from the initialization of the variable, as in the following.

```
int i;
for (i = 1; i <= 5; i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

In effect, we have taken the `int i` part out of the loop itself and put it outside the loop. This change means that the variable's scope is greater than it was before. Its scope extends to the end of the enclosing set of curly braces. One advantage of this approach is that we can refer to the final value of the control variable after the loop. Normally we wouldn't be able to do so because its scope would be limited to the loop itself. But declaring the control variable outside the loop is a dangerous practice that you will generally want to avoid. It also provides a good example of the problems you can encounter when you don't localize variables. Consider the following code, for example:

```
int i;
for (i = 1; i <= 5; i++) {
    for (i = 1; i <= 10; i++) {
        System.out.println("hi there.");
    }
}
```

As noted above, you shouldn't use the same control variable when you have nested loops. But unlike the previous example, this one actually compiles. So instead of getting a helpful error message from the Java compiler, you get a program with a bug in it. You'd think from reading these loops that it produces 50 lines of output but it actually produces just 10 lines of output. The inner loop increments the variable i until it becomes 11 and that causes the outer loop to terminate after just one iteration. It can be even worse. If you reverse the order of these loops:

```
int i;
for (i = 1; i <= 10; i++) {
    for (i = 1; i <= 5; i++) {
        System.out.println("hi there.");
    }
}
```

You get something known as an *infinite loop*.

Infinite Loop

A loop that never terminates.

This loop is infinite because no matter what the outer loop does to the variable i, the inner loop always sets it back to 1 and iterates until it becomes 6. Then the outer loop increments the variable to 7 and finds that 7 is less than or equal to 10, so it always goes back to the inner loop which once again sets the variable back to 1 and iterates up to 6. This process goes on indefinitely. These are the kinds of interference problems you can get when you fail to localize variables.

Common Programming Error: Referring to the wrong loop variable

The following code attempts to print a triangle of stars. However, it has a subtle bug that causes it to print stars infinitely.

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= i; i++) {
        System.out.print("*");
    }
    System.out.println();
}
```

The problem is on the second line, in the second for loop header's update statement. The programmer meant to write `j++` but instead accidentally wrote `i++`. A trace of the code looks like this:

```
initialization    int i = 1;    variable i is allocated and initialized to 1

initialization    int j = 1;    variable j is allocated and initialized to 1

test              j <= i        true because 1 <= 1, so we enter the inner loop
body              {...}         execute the print with j equal to 1
update            i++           increment i, which becomes 2

test              j <= i        true because 1 <= 2, so we enter the inner loop
body              {...}         execute the print with j equal to 1
update            i++           increment i, which becomes 3

...               ...           ...
```

The j variable should be increasing, but i is instead. The effect of this mistake is that the variable j is never incremented in the inner loop, and therefore the test of `j <= i` never fails, so the inner loop doesn't terminate.

Here's another broken piece of code. This one tries to print a 6x4 box of stars, but it also prints infinitely.

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; i <= 4; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

The problem is on the second line, this time in the inner for loop header's test. The programmer meant to write `j <= 4` but instead accidentally wrote `i <= 4`. Since the value of i is never incremented in the inner loop, the test of `i <= 4` never fails, so the inner loop again doesn't terminate.

Pseudocode

As we write more complex algorithms, we will find that we can't just write the entire algorithm immediately. Instead, we will increasingly make use of the technique of writing *pseudocode*.

Pseudocode

English-like descriptions of algorithms. Programming with pseudocode involves successively refining an informal description until it is easily translated into Java.

For example, you can describe the problem of drawing a box as:

```
draw a box with 50 lines and 30 columns of asterisks.
```

While this describes the figure, it is not specific about how to draw it, what algorithm to use. Do you draw the figure line-by-line or column-by-column? In Java, figures like these must be generated line by line because once a println has been performed on a line of output, that line cannot be changed. There is no command for going back to a previous line in an output file. Therefore, the first line must be output in its entirety first, then the second line in its entirety, and so on. As a result, your decompositions for these figures will be line-oriented at the top level. Thus, a closer approximation is:

```
for (each of 50 lines) {
    draw a line of 30 asterisks.
}
```

Even this can be made more specific by introducing the idea of writing a single character on the output line versus moving to a new line of output:

```
for (each of 50 lines) {
    for (each of 30 columns) {
        write one asterisk on the output line.
    }
    go to a new output line.
}
```

Using pseudocode, you can gradually convert an English description into something easily translated into a Java program. The simple examples you have seen so far are hardly worth the application of pseudocode, so we will now examine the problem of generating a more complex figure:

```
*********
 *******
  *****
   ***
    *
```

This figure must also be generated line by line:

```
for (each of 5 lines) {
    draw one line of the triangle.
}
```

Unfortunately, each line is different. Therefore, you must come up with a general rule that fits all lines. The first line of this figure has a series of asterisks on it with no leading spaces. The subsequent lines have a series of spaces followed by a series of asterisks. Using your imagination a bit, you can say that the first line has 0 spaces on it followed by a series of asterisks. This allows you to write a general rule for making this figure:

```
for (each of 5 lines) {
    write some spaces (possibly 0) on the output line.
    write some asterisks on the output line.
    go to a new output line.
}
```

In order to proceed, you must determine a rule for the number of spaces and a rule for the number of asterisks. Assuming that the lines are numbered 1 through 5 and looking at the figure, you can fill in the following chart:

Line	Spaces	Asterisks
1	0	9
2	1	7
3	2	5
4	3	3
5	4	1

You want to find a relationship between line number and the other two columns. This is simple algebra, because these columns are related in a linear way. The second column is easy to get from the line number. It equals (line - 1). The third column is a little tougher. Because it goes down by 2 every time and the first column goes up by 1 every time, you need a multiplier of -2. Then you need an appropriate constant. The number 11 seems to do the trick, so that the third column equals (11 - 2 * line). You can improve your pseudocode, then, as follows:

```
for line going 1 to 5 {
    write (line - 1) spaces on the output line.
    write (11 - 2 * line) asterisks on the output line.
    go to a new output line.
}
```

This is simple to turn into a program:

```
 1  public class DrawV {
 2      public static void main(String[] args) {
 3          for (int line = 1; line <= 5; line++) {
 4              for (int i = 1; i <= (line - 1); i++) {
 5                  System.out.print(" ");
 6              }
 7              for (int i = 1; i <= (11 - 2 * line); i++) {
 8                  System.out.print("*");
 9              }
10              System.out.println();
11          }
12      }
13  }
```

A Decrementing for Loop

Sometimes we manage complexity by taking advantage of work that we have already done. For example, how would you produce this figure?

```
    *
   ***
  *****
 *******
*********
```

You could follow the same process you did above and find new expressions that produce the appropriate number of spaces and asterisks. However, there is an easier way. This figure is the same as the previous one, except the lines appear in reverse order. We can achieve this result by running the for loop backwards. So instead of starting at 1 and going up to 5 with a ++ update, we can start at 5 and go down to 1 using a -- update.

For example, the following loop:

```
for (int i = 10; i >= 1; i--) {
    System.out.println(i + " squared = " + (i * i));
}
```

will produce the squares of the first ten integers, but in reverse order. The simple way to produce the upward-pointing triangle, then, is:

```
 1  public class DrawCone {
 2      public static void main(String[] args) {
 3          for (int line = 5; line >= 1; line--) {
 4              for (int i = 1; i <= (line - 1); i++) {
 5                  System.out.print(" ");
 6              }
 7              for (int i = 1; i <= (11 - 2 * line); i++) {
 8                  System.out.print("*");
 9              }
10              System.out.println();
11          }
12      }
13  }
```

Class Constants

The DrawCone program in the last section draws a cone with 5 lines. How would you modify it to produce a cone with 3 lines? One simple strategy is to change all the 5's to 3's, which will produce this output:

```
  *****
 *******
*********
```

This output is obviously wrong. If you work through the geometry of the figure, you will discover that the problem is with the number 11 in one of the expressions. The number 11 comes from this formula:

```
2 * (number of lines) + 1
```

Thus, for 5 lines the appropriate value is 11. But for 3 lines the appropriate value is 7. Programmers call numbers like these *magic numbers*. They are magic in the sense that they seem to make the program work, but their definition is not always obvious. Glancing at the program, one is apt to ask, "Why 5? Why 11? Why 3? Why 7? Why me?"

To make programs more readable and more adaptable, you should try to avoid magic numbers whenever possible. You do so by storing the magic numbers. You can use variables to store these values, but that is misleading, given that you are trying to represent values that don't change. Java

offers an alternative. You can declare values that are similar to variables except for the fact that they are guaranteed to have constant value. Not surprisingly, they are called *constants*. We most often define *class constants*, which can be accessed throughout the entire class.

Constant, Class Constant

A named value that cannot be changed. A class constant can be accessed anywhere in the class. (i.e., its scope is the entire class.)

The first advantage of a constant is that you can name it, which allows you to choose a descriptive name that explains what the constant represents. You can then use that name instead of referring to the specific value to make your programs more readable and adaptable. For example, in the DrawCone program you might want to introduce a constant called LINES that will replace the magic number 5 (recall from chapter 1 that we use all uppercase letters for constant names). Also, you can use the constant as part of an expression to calculate a value. This approach allows you to replace the magic number 11 with a formula like (2 * LINES + 1).

Constants are declared with the keyword `final` which indicates the fact that their values cannot be changed once assigned. You can declare them anywhere you can declare a variable, as in:

```
final int LINES = 5;
```

These values can be declared inside a method just like a variable, but they are often used by several different methods. As a result, we generally declare constants outside of methods. This requirement causes us to have another run-in with our old pal the `static` keyword. If we want to declare a constant that our static methods can access, then the constant itself has to be static. And just as we declare our methods to be public, we usually declare our constants to be public. The following is the general syntax for constant definitions:

```
public static final <type> <identifier> = <expression>;
```

For example, here are definitions for two constants.

```
public static final int HEIGHT = 10;
public static final int WIDTH = 20;
```

These definitions create constants called HEIGHT and WIDTH that will always have the values 10 and 20. These are known as class constants because we declare them in the outermost scope of the class, along with the methods of the class. That way they are visible in each method of the class.

How would you rewrite the DrawCone program with a constant to eliminate the magic numbers? You would introduce a constant for the number of lines:

```
public static final int LINES = 5;
```

Next, you replace the 5 in the outer loop with this constant. Then, you replace the 11 in the second inner loop with the expression (2 * LINES + 1). The result is the following program.

```
 1  public class DrawCone2 {
 2      public static final int LINES = 5;
 3
 4      public static void main(String[] args) {
 5          for (int line = LINES; line >= 1; line--) {
 6              for (int i = 1; i <= (line - 1); i++) {
 7                  System.out.print(" ");
 8              }
 9              for (int i = 1; i <= (2 * LINES + 1 - 2 * line); i++) {
10                  System.out.print("*");
11              }
12              System.out.println();
13          }
14      }
15  }
```

The advantage of this program is that it is more readable and more adaptable. You can make a simple change to the constant LINES to make it produce a different size figure.

2.5 Case Study: A Complex Figure

Now consider an example that is even more complex. To solve it, we will go through three basic steps:

1. Decompose the task into subtasks, each of which will become a static method.
2. For each subtask, make a table for the figure and compute formulas for each column of the table in terms of the line number.
3. Convert the tables into actual for loop code for each method.

The output we want to produce is the following.

```
+------+
|\    /|
| \  / |
|  \/  |
|  /\  |
| /  \ |
|/    \|
+------+
```

Problem Decomposition and Pseudocode

In order to generate this figure, you have to first break it down into subfigures. In doing so, you should look for lines that are similar in one way or another. The first and last lines are exactly the same. The three lines after the first line all fit one pattern, and the three lines after that fit another.

```
+------+    line

|\    /|
| \  / |    top half
|  \/  |

|  /\  |
| /  \ |    bottom half
|/    \|

+------+    line
```

Thus, you can break the overall problem down as:

```
draw a solid line.
draw the top half of the hourglass.
draw the bottom half of the hourglass.
draw a solid line.
```

You should solve each independently. Eventually we will want to incorporate a class constant to make the program more flexible, but let's first solve the problem without worrying about magic numbers.

The solid line task can be further specified as:

```
write a plus on the output line.
write 6 dashes on the output line.
write a plus on the output line.
go to a new output line.
```

This translates easily into a static method:

```
// Produces a solid line
public static void drawLine() {
    System.out.print("+");
    for (int i = 1; i <= 6; i++) {
        System.out.print("-");
    }
    System.out.println("+");
}
```

Line Pattern Table

The top half of the hourglass is more complex. Here is a typical line:

```
| \  / |
```

This has four printing characters and some spaces that separate them:

```
 |              \               /             |
bar  spaces  backslash  spaces  slash  spaces  bar
```

Thus, a first approximation in pseudocode:

```
for (each of 3 lines) {
    write a bar on the output line.
    write some spaces on the output line.
    write a back slash on the output line.
    write some spaces on the output line.
    write a slash on the output line.
    write some spaces on the output line.
    write a bar on the output line.
    go to a new line of output.
}
```

Again, you can make a table to figure out the desired expressions. Writing the single characters will be easy enough to translate into Java, but you need to be more specific about the spaces. This line really has three sets of spaces. Here is a table that shows how many to use in each case:

Line	Spaces	Spaces	Spaces
1	0	4	0
2	1	2	1
3	2	0	2

The first and third sets of spaces fit the rule (line - 1), and the second number of spaces is (6 - 2 * line). Therefore, the pseudocode should read:

```
for (line going 1 to 3) {
    write a bar on the output line.
    write (line - 1) spaces on the output line.
    write a back slash on the output line.
    write (6 - 2 * line) spaces on the output line.
    write a slash on the output line.
    write (line - 1) spaces on the output line.
    write a bar on the output line.
    go to a new line of output.
}
```

Initial Structured Version

Our pseudocode for the top half of the hourglass is easily translated into a static method named drawTop. A similar solution exists for the bottom half of the hourglass. Put together, the program looks like this:

```
1  public class DrawFigure {
2      public static void main(String[] args) {
3          drawLine();
4          drawTop();
5          drawBottom();
6          drawLine();
7      }
8
9      // Produces a solid line
10     public static void drawLine() {
11         System.out.print("+");
12         for (int i = 1; i <= 6; i++) {
13             System.out.print("-");
14         }
15         System.out.println("+");
16     }
17
18     // This produces the top half of the hourglass figure
19     public static void drawTop() {
20         for (int line = 1; line <= 3; line++) {
21             System.out.print("|");
22             for (int i = 1; i <= (line - 1); i++) {
23                 System.out.print(" ");
24             }
25             System.out.print("\\");
26             for (int i = 1; i <= (6 - 2 * line); i++) {
27                 System.out.print(" ");
28             }
29             System.out.print("/");
30             for (int i = 1; i <= (line - 1); i++) {
31                 System.out.print(" ");
32             }
33             System.out.println("|");
34         }
35     }
36
37     // This produces the bottom half of the hourglass figure
38     public static void drawBottom() {
39         for (int line = 1; line <= 3; line++) {
40             System.out.print("|");
41             for (int i = 1; i <= (3 - line); i++) {
42                 System.out.print(" ");
43             }
44             System.out.print("/");
45             for (int i = 1; i <= 2 * (line - 1); i++) {
46                 System.out.print(" ");
47             }
48             System.out.print("\\");
49             for (int i = 1; i <= (3 - line); i++) {
50                 System.out.print(" ");
51             }
52             System.out.println("|");
53         }
54     }
55 }
```

Adding a Class Constant

The DrawFigure program works in that it produces the output we started with, but it is not very flexible. What if we wanted to produce a similar figure of a different size? The original problem involved an hourglass figure that had 3 lines in the top half and three lines in the bottom half. What if we wanted the following output, with 4 lines in the top half and 4 lines in the bottom half?

```
+--------+
|\      /|
| \    / |
|  \  /  |
|   \/   |
|   /\   |
|  /  \  |
| /    \ |
|/      \|
+--------+
```

Obviously the program would be more useful if we could make it flexible enough to produce either output. We do so by eliminating the magic numbers with the introduction of a class constant. You might think we could introduce two constants for the height and width, but because of the regularity of this figure, the height is determined by the width and vice versa.

So we want to introduce a single class constant. Let's use the height of the hourglass halves:

```
public static final int SUB_HEIGHT = 4;
```

We refer to it as "sub height" rather than "height" because this is the height of each of the two halves. Notice how we use the underscore character to separate the different words in the name of the constant.

So how do we modify the original program to incorporate this constant? We look through it for any magic numbers and modify when appropriate. For example, the drawLine method draws 6 dashes for the subheight of 3. If you look at the new figure, you will see that it has 8 dashes for a subheight of 4. You can use the same kind of reasoning that we used with our tables. If the subheight is going up by 1 and the number of dashes goes up by 2, then we're going to need a multiplier of 2. So the number of dashes will involve (2 * SUB_HEIGHT). We also might need a constant, although not in this case because the expression gives us exactly what we are looking for (6 dashes for a subheight of 3, 8 dashes for a subheight of 4).

So the drawLine method would be rewritten as follows:

```
// Produces a solid line
public static void drawLine() {
    System.out.print("+");
    for (int i = 1; i <= (2 * SUB_HEIGHT); i++) {
        System.out.print("-");
    }
    System.out.println("+");
}
```

There are quite a few magic numbers in the drawTop and drawBottom methods. In particular, the numbers 1, 2, 3 and 6 appear several times. In some cases it is fairly obvious what to do. For example, each method has an outer for loop that uses the magic number 3. Why 3? Because each subfigure has a height of 3. Obviously that 3 should be replaced with SUB_HEIGHT. The number 6 is not quite so obvious. You could make an educated guess that if the magic number is 6 when the subheight is 3, then perhaps it's equal to twice the subheight. In fact, that turns out to be the right answer.

If guessing doesn't work, you can always fall back on the table technique. Work out a new table for the figure of subheight 4 and figure out what expressions to use for it. If you do so, you'll find that the magic number 6 in the old expressions is replaced by the magic number 8. If the magic number needs to go up by 2 when the subheight goes up by 1, clearly we need a multiplier of 2.

The numbers 1 and 2 turn out not to be related to the subheight. Again, this is something you can make an educated guess about and verify by executing the program or you can work out a new set of formulas to see whether these numbers change with a new subheight.

Here is the new version of the program with a class constant for the subheight. It uses a subheight value of 4, but we could change this number to 3 to get the smaller version and we could change it to some other number to get yet another version of the figure.

```
1  public class DrawFigure2 {
2      public static final int SUB_HEIGHT = 4;
3
4      public static void main(String[] args) {
5          drawLine();
6          drawTop();
7          drawBottom();
8          drawLine();
9      }
10
11     // Produces a solid line
12     public static void drawLine() {
13         System.out.print("+");
14         for (int i = 1; i <= (2 * SUB_HEIGHT); i++) {
15             System.out.print("-");
16         }
17         System.out.println("+");
18     }
19
20     // This produces the top half of the hourglass figure
21     public static void drawTop() {
22         for (int line = 1; line <= SUB_HEIGHT; line++) {
23             System.out.print("|");
24             for (int i = 1; i <= (line - 1); i++) {
25                 System.out.print(" ");
26             }
27             System.out.print("\\");
28             for (int i = 1; i <= (2 * SUB_HEIGHT - 2 * line); i++) {
29                 System.out.print(" ");
30             }
31             System.out.print("/");
32             for (int i = 1; i <= (line - 1); i++) {
33                 System.out.print(" ");
34             }
```

```
35              System.out.println("|");
36          }
37      }
38
39      // This produces the bottom half of the hourglass figure
40      public static void drawBottom() {
41          for (int line = 1; line <= SUB_HEIGHT; line++) {
42              System.out.print("|");
43              for (int i = 1; i <= (SUB_HEIGHT - line); i++) {
44                  System.out.print(" ");
45              }
46              System.out.print("/");
47              for (int i = 1; i <= 2 * (line - 1); i++) {
48                  System.out.print(" ");
49              }
50              System.out.print("\\");
51              for (int i = 1; i <= (SUB_HEIGHT - line); i++) {
52                  System.out.print(" ");
53              }
54              System.out.println("|");
55          }
56      }
57  }
```

This program raises an important issue. It declares a constant called SUB_HEIGHT that is used throughout the program. But the constant is not declared locally in the individual methods. That seems to violate our principle of localizing whenever possible. While localizing of variables is a good idea, the same, is not always true for constants. In this case a class-wide definition is more appropriate. We localize variables to avoid potential interference. That argument doesn't hold for constants, since they are guaranteed not to change. Another argument for using local variables is that it makes our static methods more independent. That argument has some merit, but not enough. It is true that class constants introduce dependencies between methods, but often that is really what you want. For example, the three methods of the hourglass program should not be independent of each other when it comes to the size of figures. Each subfigure has to use the same size constant. Imagine the potential disaster if each method had its own SUB_HEIGHT each with a different value. None of the pieces would fit together.

Further Variations

This solution may seem cumbersome, but it is easier to adapt to a new task. For example, suppose that you want to generate the following output:

```
+----------+
|\        /|
| \      / |
|  \    /  |
|   \  /   |
|    \/    |
|    /\    |
|   /  \   |
|  /    \  |
| /      \ |
|/        \|
+----------+
|    /\    |
|   /  \   |
|  /    \  |
| /      \ |
|/        \|
|\        /|
| \      / |
|  \    /  |
|   \  /   |
|    \/    |
+----------+
```

This output uses a subheight of 5 and includes both a diamond pattern and an "X" pattern. We can produce this output by changing the subheight constant to 5:

```
public static final int SUB_HEIGHT = 4;
```

And rewriting the main method as follows to produce both the original diamond pattern and the new "X" pattern which we get simply by reversing the order of the calls on the two halves:

```
public static void main(String[] args) {
    drawLine();
    drawTop();
    drawBottom();
    drawLine();
    drawBottom();
    drawTop();
    drawLine();
}
```

Chapter Summary

- Java groups data into types. There are two major categories of data types: primitive data and objects. Primitive types include int (integers), double (real numbers), char (individual text characters), and boolean (logical values).
- Values and computations are called expressions. The simplest expressions are individual values, also called literals. Some example literals are: `42`, `3.14`, `'Q'`, and `false`. Expressions may contain operators, such as: `(3 + 29) - 4 * 5`. The division operation is odd in that it's split into quotient (/) and remainder (%) operations.

- Rules of precedence determine the order in multiple operators are evaluated in complex expressions. In particular, multiplication and division are performed before addition and subtraction. Parentheses can be used to force a particular order of evaluation.
- Data can be converted from one type to another by an operation called a cast.
- Variables are memory locations in which a value can be stored. A variable is declared with a name and a type. Any data value with a compatible type can be stored in the variable's memory and used later in the program.
- Primitive data can be printed on the console using the System.out.println method, just like text strings. A String can be connected with another value ("concatenated") with the + operator to produce a larger string. This allows you to print complex expressions with text on the console.
- A loop is used to execute a group of statements several times. The for loop is one kind of loop that can be used to apply the same statements over a range of numbers, or to repeat statements a determined number of times.
- A loop can contain another loop, also called a nested loop.
- The System.out.print statement writes a partial line of output to the console without advancing to the next line. System.out.print is useful when writing for loops that produce many parts of a line of output.
- A variable exists from the line it is declared to the end of the curly braces in which it was declared. This range, also called the scope of the variable, constitutes the legal part of the program where the variable can legally be used.
- A variable declared inside a method or loop is called a local variable. A local variable can only be used inside its method or loop.
- An algorithm can be easier to write if you first write an English description of it. Such a description is also called pseudo-code.
- Important constant values written into the program should instead be declared as class constants, both to explain their name and value and to make it easier to change their value later.

Self-Check Problems

Section 2.1: Basic Data Concepts

1. Which of the following are legal int literals?

```
22     1.5     -1     2.3     10.0     5.   -6875309     '7'
```

2. Trace the evaluation of the following expressions, and give their resulting values:
   - `2 + 3 * 4 - 6`
   - `14 / 7 * 2 + 30 / 5 + 1`
   - `(12 + 3) / 4 * 2`
   - `(238 % 10 + 3) % 7`
   - `12 - 2 - 3`
   - `6/2 + 7/3`
   - `6 * 7%4`
   - `3 * 4 + 2 * 3`

- `(18 - 7) * (43 % 10)`
- `2 + 19 % 5 - (11 * (5 / 2))`
- `813 % 100/3 + 2.4`
- `26 % 10 % 4 * 3`
- `22 + 4 * 2`
- `23 % 8 % 3`
- `177 % 100 % 10/2`
- `89 % (5 + 5) % 5`
- `392/10 % 10/2`
- `8 * 2 - 7/4`
- `37 % 20 % 3 * 4`

3. Trace the evaluation of the following expressions, and give their resulting values:
   - `17 % 10/4`
   - `4.0/2 * 9/2`
   - `2.5 * 2 + 8/5.0 + 10/3`
   - `12/7 * 4.4 * 2/4`
   - `4 * 3/8 + 2.5 * 2`
   - `(5 * 7.0/2 - 2.5)/5 * 2`
   - `12/7 * 4.4 * 2/4`
   - `41 % 7 * 3/5 + 5/2 * 2.5`
   - `10.0/2/4`
   - `8/5 + 13/2/3.0`
   - `(2.5 + 3.5)/2`
   - `9/4 * 2.0 - 5/4`
   - `9/2.0 + 7/3 - 3.0/2`
   - `813 % 100/3 + 2.4`
   - `27/2/2.0 * (4.3 + 1.7) - 8/3`
   - `4.0/2 * 9/2`
   - `53/5/(0.6 + 1.4)/2 + 13/2`
   - `2.5 * 2 + 8/5.0 + 10/3`
   - `2 * 3/4 * 2/4.0 + 4.5 - 1`
   - `89 % 10/4 * 2.0/5 + (1.5 + + 1.0/2) * 2`

4. Trace the evaluation of the following expressions, and give their resulting values:
   - `2 + 2 + 3 + 4`
   - `"2 + 2 " + 3 + 4`
   - `2 + " 2 + 3 " + 4`
   - `3 + 4 + " 2 + 2"`
   - `"2 + 2 " + (3 + 4)`
   - `"(2 + 2) " + (3 + 4)`
   - `"hello 34 " + 2 * 4`

Section 2.2: Variables

5. Imagine you are writing a personal fitness program that stores the user's age, gender, height (in feet or meters) and weight (to the nearest pound or kilogram). Declare variables with the appropriate names and types to hold this information.

6. Imagine you are writing a student program that stores the student's year (Freshman, Sophomore, Junior, or Senior), number of courses, and GPA on a 4.0 scale. Declare variables with the appropriate names and types to hold this information.

7. Suppose you have an int variable named `number`. What Java expression produces the last digit of the number (the 1s place)?

8. Suppose you have an int variable named `number`. What Java expression produces the second to last digit of the number (the 10s place)? What expression produces the third to last digit of the number (the 100s place)?

9. Consider the following code:

```
int first = 8;
int second = 19;
first = first + second;
second = first - second;
first = first - second;
```

What are the values of `first` and `second` at the end of this code? How would you describe the net effect of the code statements in this exercise?

10. Rewrite the code from the previous exercise to be shorter, by declaring the variables together and by using the special assignment operators such as +=, -=, *= and /= as appropriate.

11. What are the values of `a`, `b`, and `c` after the following code statements? (It may help you to write down their values after each line.)

```
int a = 5;
int b = 10;
int c = a++ + ++b;
a++;
b--;
```

Section 2.3: The for Loop

12. Assume that you have a variable named `count` that will take on the values 1, 2, 3, 4, and so on. You are going to formulate expressions in terms of count that will yield different sequences. For example, to get the sequence 2, 4, 6, 8, 10, 12, ..., you would use the expression `(2 * count)`. Fill in the following table, indicating an expression that will generate each sequence.

Sequence	Expression
4, 19, 34, 49, 64, 79, ...	
30, 20, 10, 0, -10, -20, ...	
-7, -3, 1, 5, 9, 13, ...	
97, 94, 91, 88, 85, 82, ...	

13. Complete the code for the following for loop:

```
for (int i = 1; i < 6; i++) {
    // your code here

}
```

so that it prints the following numbers, one per line:

```
-4
14
32
50
68
86
```

14. Rewrite the following code as a series of equivalent System.out.println statements (i.e., without any System.out.print statements):

```
System.out.print("Twas ");
System.out.print("brillig and the");
System.out.println(" ");
System.out.print("   slithy toves did");
System.out.print(" ");
System.out.println("gyre and");
System.out.println(   "gimble");
System.out.println();
System.out.println(   "in the wabe."  );
```

15. Suppose that you are trying to write a program that produces the following output:

```
1 3 5 7 9 11 13 15 17 19 21
1 3 5 7 9 11
```

The following program is an attempt at a solution, but it contains four major errors. Identify them all.

```
 1  public class BadNews {
 2      public static final int MAX_ODD = 21;
 3
 4      public static void writeOdds() {
 5          // print each odd number
 6          for (int count = 1; count <= (MAX_ODD - 2); count++) {
 7              System.out.print(count + " ");
 8              count = count + 2;
 9          }
10
11          // print the last odd number
12          System.out.print(count + 2);
13      }
14
15      public static void main(String[] args) {
16          // write all odds up to 21
17          writeOdds();
18
19          // now, write all odds up to 11
20          MAX_ODD = 11;
21          writeOdds();
22      }
23  }
```

16. What is the output of the following method `oddStuff`:

```
public static void oddStuff() {
    int number = 4;

    for (int count = 1; count <= number; count++) {
        System.out.println(number);
        number = number / 2;
    }
}
```

17. What is the output of the following method `unknown`:

```
 1  public class Strange {
 2      public static final int MAX = 5;
 3
 4      public static void unknown() {
 5          int number = 0;
 6
 7          for (int count = MAX; count >= 1; count--) {
 8              number += (count * count);
 9          }
10
11          System.out.println("The result is: " + number);
12      }
13
14      public static void main(String[] args) {
15          unknown();
16      }
17  }
```

18. What is the output of the following loop?

```
int total = 25;
for (int number = 1; number <= (total / 2); number++) {
    total = total - number;
    System.out.println(total + "  " + number);
}
```

19. What is the output of the following loop?

```
System.out.println("+----+");
for (int i = 1; i <= 3; i++) {
    System.out.println("\\    /");
    System.out.println("/    \\");
}
System.out.println("+----+");
```

20. What is the output of the following loop?

```
for (int i = 1; i <= 3; i++)
    System.out.println("How many lines");
    System.out.println("are printed?");
```

21. What is the output of the following loop?

```
System.out.print("T-minus ");
for (int i = 5; i >= 1; i--) {
     System.out.print(i + ", ");
}
System.out.println("Blastoff!");
```

22. What is the output of the following sequence of loops?

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.print((i * j) + " ");
    }
    System.out.println();
}
```

23. What is the output of the following sequence of loops?

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10 - i; j++) {
        System.out.print(" ");
    }
    for (int j = 1; j <= 2 * i - 1; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

24. What is the output of the following sequence of loops?

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        for (int k = 1; k <= 4; k++) {
            System.out.print("*");
        }
        System.out.print("!");
    }
    System.out.println();
}
```

25. What is the output of the following sequence of loops? Notice that it is the same as the previous exercise, except that the placement of the braces has changed.

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        for (int k = 1; k <= 4; k++) {
            System.out.print("*");
        }
    }
    System.out.print("!");
    System.out.println();
}
```

26. What is the output of the following sequence of loops? Notice that it is the same as the previous exercise, except that the placement of the braces has changed.

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        for (int k = 1; k <= 4; k++) {
            System.out.print("*");
            System.out.print("!");
        }
        System.out.println();
    }
}
```

Section 2.4: Managing Complexity

27. Write a Java program that produces the following output. Use nested for loops to capture the structure of the figure.

```
!!!!!!!!!!!!!!!!!!!!
\\!!!!!!!!!!!!!!!!!!//
\\\\!!!!!!!!!!!!!!!!////
\\\\\\!!!!!!!!!!!!//////
\\\\\\\\!!!!!!!!!!////////
\\\\\\\\\\!!!!!!!!!!
```

28. Modify your program from the previous exercise so that it uses a global constant for the figure's height. The previous output used a constant height of 6. Here is the output for a constant height of 4:

```
!!!!!!!!!!!!!!
\\!!!!!!!!!!!!//
\\\\!!!!!!!!////
\\\\\\!!!!!!//////
```

Exercises

1. In physics, a common useful equation for finding the position s of a body in linear motion at a given time t, based on its initial position s_0, initial velocity v_0, and rate of acceleration a is the following:

$$s = s_0 + v_0 t + \tfrac{1}{2} a t^2$$

   Write code to declare variables for s_0, v_0, a, and t, and then write the code to compute s based on these values.

2. Write a for loop that produces the following output:

```
1 4 9 16 25 36 49 64 81 100
```

   For added challenge, try to modify your code from the previous exercise so that it does not need to use the * multiplication operator. (It can be done! Hint: Look at the differences between adjacent numbers.)

3. The Fibonacci numbers are a sequence of integers where the first two elements are 1, and each following element is the sum of the two preceding elements. The mathematical definition of each kth Fibonacci is the following:

$$F(k)\begin{cases} F(k-1)+F(k-2), k>2 \\ 1, k \le 2 \end{cases}$$

   The first 12 Fibonacci numbers are:

```
1 1 2 3 5 8 13 21 34 55 89 144
```

   Write a for loop that computes and prints the first 12 Fibonacci numbers.

4. Write for loops to produce the following output:

```
*****
*****
*****
*****
```

5. Write for loops to produce the following output:

```
*
**
***
****
*****
```

6. Write for loops to produce the following output:

```
1
22
333
4444
55555
666666
7777777
```

7. Write for loops to produce the following output:

```
    1
   2
  3
 4
5
```

8. Write for loops to produce the following output:

```
    1
   22
  333
 4444
55555
```

9. Write for loops to produce the following output, with each line 40 characters wide:

```
----------------------------------------
_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-
1122334455667788990011223344556677889900
----------------------------------------
```

10. It's common to print a rotating, increasing list of single-digit numbers at the start of a program's output as a visual guide to number the columns of the output to follow. With this in mind, write nested for loops to produce the following output, with each line 60 characters wide:

```
         |         |         |         |         |         |
123456789012345678901234567890123456789012345678901234567890
```

11. Modify your code from the previous exercise so that it could easily be modified to display a different range of numbers (instead of 1234567890), a different number of repetitions of those numbers (instead of 60 total characters) and keep the vertical bars matching up correctly. Use class constants instead of "magic numbers." Example outputs that could be generated by changing your constants would be:

```
    |    |    |    |    |    |    |    |    |    |
12340123401234012340123401234012340123401234012340

       |       |       |       |       |       |       |
12345670123456701234567012345670123456701234567012345670
```

12. Write nested for loops that produce the following output. Indicate the level of each statement in your code using indentation and a comment.

```
000111222333444555666777888999
000111222333444555666777888999
000111222333444555666777888999
```

13. Modify the code so that it now produces the following output:

```
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
```

14. Modify the code so that it now produces the following output:

```
99999999988888888777777766666655555444433221
99999999988888888777777766666655555444433221
99999999988888888777777766666655555444433221
99999999988888888777777766666655555444433221
```

15. Write a method named `printDesign` that produces the following output. Use for loops to capture the structure of the figure.

```
-----1-----
----333----
---55555---
--7777777--
-999999999-
```

16. Write a pseudo-code algorithm that will produce the following figure as output.

```
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
```

17. Use your pseudo-code from the previous exercise to write a Java program that produces the preceding figure as output. Use for loops to print the repeated parts of the figure. Once you get it to work, add a class constant so that the size of the figure could be changed simply by changing the constant's value.

Programming Projects

1. Write a program that produces the following output:

```
****** /////////// ******
*****  //////////\  *****
****   ////////\\\\   ****
***    //////\\\\\\    ***
**     ////\\\\\\\\     **
*      //\\\\\\\\\\      *
       \\\\\\\\\\\\
```

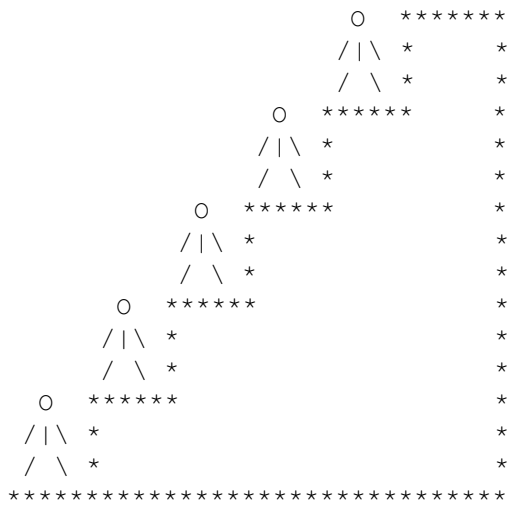
2. Write a program that produces the following output:

```
+------+
|  ^^  |
| ^  ^ |
|^    ^|
|  ^^  |
| ^  ^ |
|^    ^|
+------+
|v    v|
| v  v |
|  vv  |
|v    v|
| v  v |
|  vv  |
+------+
```

3. Write a program that produces the following output:

```
+---------+
|    *    |
|   /*\   |
|  //*\\  |
| ///*\\\ |
| \\\*/// |
|  \\*//  |
|   \*/   |
|    *    |
+---------+
| \\\*/// |
|  \\*//  |
|   \*/   |
|    *    |
|    *    |
|   /*\   |
|  //*\\  |
| ///*\\\ |
+---------+
```

4. Write a program that produces the following output. Use a class constant to make it possible to change the number of stairs in the figure.



5. Write a program that produces the following rocket ship figure as its output. Use a class constant to make it possible to change the size of the rocket (the following output uses a size of 3).

```
     /**\
    //**\\
   ///**\\\
  ////**\\\\
 /////**\\\\\
+=*=*=*=*=*=*=*+
|../\..../\..|
|./\/\../\/\.|
|/\/\/\/\/\/\|
|\/\/\/\/\/\/|
|.\/\/..\/\/.|
|..\/....\/..|
+=*=*=*=*=*=*=*+
|\/\/\/\/\/\/|
|.\/\/..\/\/.|
|..\/....\/..|
|../\..../\..|
|./\/\../\/\.|
|/\/\/\/\/\/\|
+=*=*=*=*=*=*=*+
     /**\
    //**\\
   ///**\\\
  ////**\\\\
 /////**\\\\\
```

Stuart Reges
Marty Stepp

Chapter 3
Introduction to Parameters and Objects

Copyright © 2006 by Stuart Reges and Marty Stepp

- 3.1 Parameters
  - The Mechanics of Parameters
  - Limitations of Parameters
  - Multiple Parameters
  - Parameters Versus Constants
  - Overloading of Methods
- 3.2 Methods that Return Values
  - The Math Class
  - Defining Methods that Return Values
- 3.3 Using Objects
  - String Objects
  - Point Objects
  - Reference Semantics
  - Multiple Objects
  - Objects as Parameters to Methods
- 3.4 Interactive Programs
  - Scanner Objects
  - A Sample Interactive Program
- 3.5 Case Study: Projectile Trajectory
  - An Unstructured Solution
  - A Structured Solution

Introduction

Chapter 2 introduced techniques for managing complexity including the use of class constants, which make programs more flexible. This chapter explores a more powerful technique for obtaining such flexibility. You will learn how to use parameters to create methods that solve not just single tasks, but whole families of tasks. Creating such methods requires an insight into problems called generalization. It requires looking beyond a specific task to find a more general task for which this task is just one example. The ability to generalize is one of the most important qualities of a software engineer, and the generalization technique you will study in this chapter is one of the most powerful techniques used by programmers.

After exploring parameters, the chapter discusses other issues associated with methods, particularly the ability of a method to return a value. The chapter ends with an introduction to using objects.

3.1 Parameters

Humans are very good at learning new tasks. In doing so, we often group a family of related tasks into one generalized solution. For example, someone might ask you to take 10 steps forward or someone might ask you to take 20 steps forward. These are different tasks, but they both involve taking a certain number of steps forward. We think of this action as a single task of taking steps forward, but we understand that the number of steps will vary from one task to another. In programming we refer to the number of steps as a parameter that allows you to generalize the task.

Parameter (parameterize)

Any of a set of characteristics that distinguish different members of a family of tasks. To parameterize a task is to identify a set of its parameters.

For a programming example, let's return to the DrawFigure program of Chapter 2. It performs its task adequately, but there are several aspects that can be improved. For example, there are six different places where a for loop writes out spaces. This approach is redundant and can be consolidated into a single method that performs all space writing tasks.

Each space writing task requires a different number of spaces, so we need some way to tell the method how many spaces to write. The methods we have been writing have a simple calling mechanism where we say:

```
writeSpaces();
```

You could imagine setting a variable to a particular value before the method is called, as in:

```
int number = 10;
writeSpaces();
```

Then the method could look at the value of the variable number to see how many spaces to write. Unfortunately, this approach won't work. Recall from Chapter 2 that scope rules determine where variables can be accessed. Following those rules, the variable number would be a local variable in main that could not be seen inside method writeSpaces.

We want to be able to somehow include this value in the call, so that if we want to get 10 spaces we say:

```
writeSpaces(10);
```

and if we want to get 20 spaces we say:

```
writeSpaces(20);
```

Parameters in Java allow us to do exactly this. They allow us to specify one or more parameters to the method. The idea is that instead of writing a method that performs just one version of a task, we write a more flexible version that solves a family of related tasks that all differ by one or more parameters. In the case of the writeSpaces method, the parameter is the number of spaces to write. This number characterizes the different space writing tasks.

The following is the definition of writeSpaces with a parameter for the number of spaces to write.

```
public static void writeSpaces(int number) {
    for (int i = 1; i <= number; i++) {
        System.out.print(" ");
    }
}
```

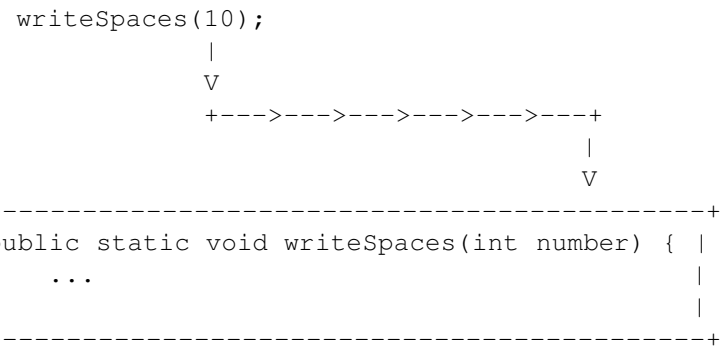
The parameter appears in the method header after the name and inside the parentheses we have been leaving empty. This method uses a parameter called number of type int. As indicated above, you can no longer call the parameterized method by using just its name:

```
writeSpaces();
```

You must now say something like:

```
writeSpaces(10);
```

When a call like this is made, the value 10 is used to initialize the parameter called number. You can think of this as information flowing into the method from the call:



The parameter number is a local variable, but it gets its initial value from the call. Given that we are calling this method with the value 10, it's as if we had included the following declaration at the beginning of method writeSpaces:

```
int number = 10;
```

Of course, this mechanism is more flexible than a specific variable declaration, because we can instead say:

```
writeSpaces(20);
```

and it will be as if we had said:

```
int number = 20;
```

at the beginning of the method. We can even use an integer expression for the call:

```
writeSpaces(3 * 4 - 5);
```

In this case Java evaluates the expression to get the value 7 and then calls writeSpaces initializing number to 7.

Computer scientists use the word parameter liberally to mean both what appears in the method header (the formal parameter) and what appears in the method call (the actual parameter).

Formal Parameter

A variable that appears inside parentheses in the header of a method that is used to generalize the method's behavior.

Actual Parameter

A specific value or expression that appears inside parentheses in a method call.

The term formal parameter is not very descriptive of its purpose. A better name would be "generalized parameter." In the method above, number is the generalized parameter that appears in the method declaration. It is a placeholder for some unspecified value. The values appearing in the method calls are the actual parameters, because each call indicates a specific task to perform. In other words, each call provides an actual value to fill the placeholder.

The word "argument" is often used as a synonym for "parameter," as in, "These are the arguments I'm passing to this method." Some people prefer to reserve the word "argument" for actual parameters and the word "parameter" for formal parameters.

Let's look at an example of how you might use this writeSpaces method. Remember that the DrawFigure2 program had the following method called drawTop:

```
// This produces the top half of the hourglass figure
public static void drawTop() {
    for (int line = 1; line <= SUB_HEIGHT; line++) {
        System.out.print("|");
        for (int i = 1; i <= (line - 1); i++) {
            System.out.print(" ");
        }
        System.out.print("\\");
        for (int i = 1; i <= (2 * SUB_HEIGHT - 2 * line); i++) {
            System.out.print(" ");
        }
        System.out.print("/");
        for (int i = 1; i <= (line - 1); i++) {
            System.out.print(" ");
        }
        System.out.println("|");
    }
}
```

Using the writeSpaces method we can rewrite this as follows:

```
public static void drawTop() {
    for (int line = 1; line <= SUB_HEIGHT; line++) {
        System.out.print("|");
        writeSpaces(line - 1);
        System.out.print("\\");
        writeSpaces(2 * SUB_HEIGHT - 2 * line);
        System.out.print("/");
        writeSpaces(line - 1);
        System.out.println("|");
    }
}
```

Notice that we call writeSpaces three different times, specifying how many spaces we want in each case. Similarly, we could also modify the drawBottom method from the DrawFigure program to simplify it.

The Mechanics of Parameters

When Java executes a call on a method, it initializes the parameters of the method. For each parameter it first evaluates the expression passed as the actual parameter and uses the result to initialize a local variable whose name is given by the formal parameter. Let's use an example to understand this process better.

```
 1  public class ParameterExample {
 2      public static void main(String[] args) {
 3          int spaces1 = 3;
 4          int spaces2 = 5;
 5
 6          System.out.print("*");
 7          writeSpaces(spaces1);
 8          System.out.println("*");
 9
10          System.out.print("!");
11          writeSpaces(spaces2);
12          System.out.println("!");
13
14          System.out.print("'");
15          writeSpaces(8);
16          System.out.println("'");
17
18          System.out.print("<");
19          writeSpaces(spaces1 * spaces2 - 5);
20          System.out.println(">");
21      }
22
23      // Writes "number" spaces on the current output line to System.out
24      public static void writeSpaces(int number) {
25          for(int i = 1;  i <= number; i++) {
26              System.out.print(" ");
27          }
28      }
29  }
```

In the first two lines of method main the computer finds instructions to allocate and initialize two variables:

```
         +---+              +---+
spaces1  | 3 |      spaces2 | 5 |
         +---+              +---+
```

The next three lines of code:

```
System.out.print("*");
writeSpaces(spaces1);
System.out.println("*");
```

produce an output line with 3 spaces bounded by asterisks on either side. You can see where the asterisks come from, but look at the method call that produces the spaces. When Java executes the call on method writeSpaces, it must set up its parameter. To set up the parameter, Java first evaluates the expression being passed as the actual parameter. The expression is simply the variable spaces1, which has the value 3. Therefore, the expression evaluates to 3. Java uses this result to initialize a local variable called number.

The following diagram indicates how memory would look as we enter method writeSpaces the first time. Because there are two methods involved (main and writeSpaces), the diagram indicates which variables are local to main (spaces1 and spaces2) and which are local to writeSpaces (the parameter number).

```
  method main                            method writeSpaces
+-----------------------------------+   +---------------------+
|          +---+              +---+ |   |         +---+       |
| spaces1  | 3 |      spaces2 | 5 | |   | number  | 3 |       |
|          +---+              +---+ |   |         +---+       |
+-----------------------------------+   +---------------------+
```

The net effect of this process is that method writeSpaces has a local copy of the value stored in the variable spaces1 from the main method. The println that comes after the call on writeSpaces puts an asterisk at the end of the line and then completes the line of output.

Let's now trace the next three lines of code:

```
System.out.print("!");
writeSpaces(spaces2);
System.out.println("!");
```

We first print an exclamation mark on the second line of output. Then we call writeSpaces again, this time with the variable spaces2 as its actual parameter. The computer evaluates this expression obtaining the result 5. This value is used to initialize number. Thus, this time it creates a copy of the value stored in the variable spaces2 from method main:

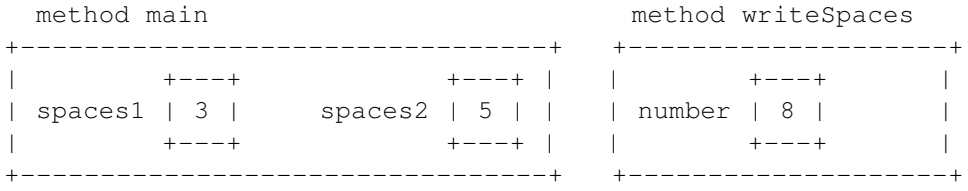
```
  method main                            method writeSpaces
+-----------------------------------+   +---------------------+
|          +---+              +---+ |   |         +---+       |
| spaces1  | 3 |      spaces2 | 5 | |   | number  | 5 |       |
|          +---+              +---+ |   |         +---+       |
+-----------------------------------+   +---------------------+
```

Because number has a different value this time (5 instead of 3), the method produces a different number of spaces. After the method executes, thed println finishes the line of output with a second exclamation mark.

Next we come across these three lines of code:

```
System.out.print("'");
writeSpaces(8);
System.out.println("'");
```

This code writes a single quotation mark at the beginning of the third line of output and then calls method writeSpaces again. This time it uses the integer literal 8 as the expression, which means it initializes the parameter number as a copy of the number 8:



Again, the method will behave differently because of the different value of number. It prints 8 spaces on line and finishes executing. Then the println completes the line of output by printing another single quotation mark at the end of the line.

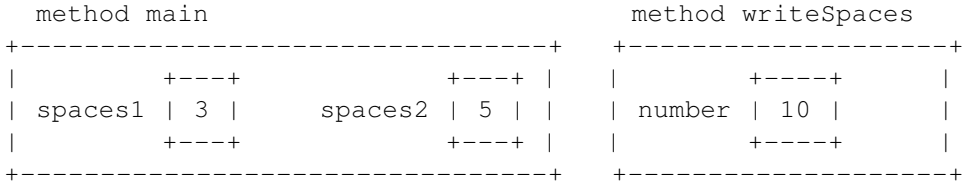
Finally we encounter these lines of code:

```
System.out.print("<");
writeSpaces(spaces1 * spaces2 - 5);
System.out.println(">");
```

The code prints a less-than character at beginning of the fourth line of output and then makes a call on the writeSpaces method. This time the actual parameter is an expression, not just a variable or literal value. That means that before the call is made, the computer evaluates the expression to determine its value:

```
spaces1 * spaces2 - 5
\----/    \-----/
  3     *    5    - 5
  \----------/
       15         - 5
       \------------/
             10
```

The computer uses this result to initialize number:



Thus, now number is a copy of the value described by this complex expression. Therefore, the total output of this program is:

```
*    *
!      !
'         '
<           >
```

Common Programming Error: Confusing Actual and Formal Parameters

Many students get used to seeing declarations of formal parameters and mistakenly believe that this is identical to the syntax for passing actual parameters. It's a common mistake to write the type of a variable as it's being passed to a parameter:

```
writeSpaces(int spaces1);   // this doesn't work
```

This confusion is often due to the fact that parameters' types are written in the declaration of the method, like this:

```
public static void writeSpaces(int number)
```

Types must be written when variables or parameters are declared. But when variables are used, such as when calling a method and passing them as actual parameters, their types are not written. Actual parameters are not declarations, so therefore types should not be written before them.

```
writeSpaces(spaces1);   // much better!
```

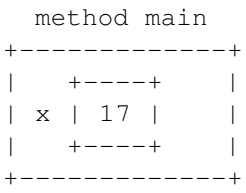
Limitations of Parameters

We've seen that a parameter can be used to provide input to a method. You can use a parameter to send a value into a method. Unfortunately, you can't use a parameter to get a value out of a method. Let's see why.

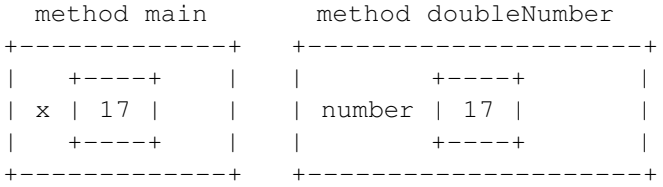
When a parameter is set up, a local variable is created and is initialized to the value being passed as the actual parameter. The net effect is that the local variable is a copy of the value coming from the outside. Since it is a local variable, it can't influence any variables outside the method. Consider the following sample program.

```
 1  public class ParameterExample2 {
 2      public static void main(String[] args) {
 3          int x = 17;
 4          doubleNumber(x);
 5          System.out.println("x = " + x);
 6          System.out.println();
 7
 8          int number = 42;
 9          doubleNumber(number);
10          System.out.println("number = " + number);
11      }
12
13      public static void doubleNumber(int number) {
14          System.out.println("Initial value of number = " + number);
15          number *= 2;
16          System.out.println("Final value of number = " + number);
17      }
18  }
```

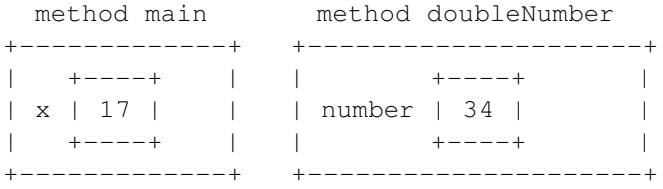
This program begins by declaring and initializing an integer variable called x with value 17.



It then calls the method doubleNumber, passing x as a parameter. The value of x is used to initialize the parameter number as a local variable of the method called doubleNumber:

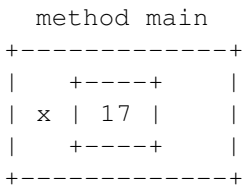


We then execute the statements inside of doubleNumber. It begins by printing the initial value of number (17). Then it doubles number:

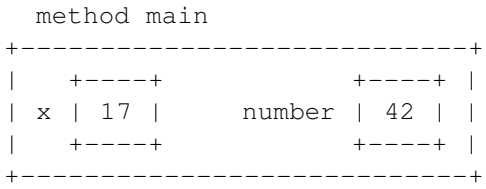


Notice that this has no effect on the variable x. The parameter called number is a copy of x, so even though they started out the same, the change to number has no effect on x. Then we report the new value of number (34).

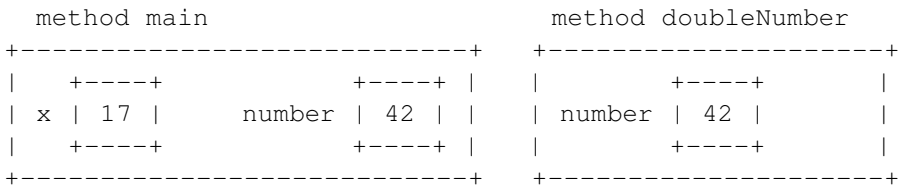
At this point, method doubleNumber finishes executing and we return to main.



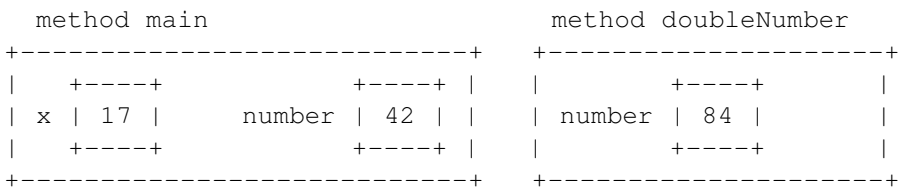
We report the value of x, which is 17. Then we declare and initialize a variable called number with value 42:



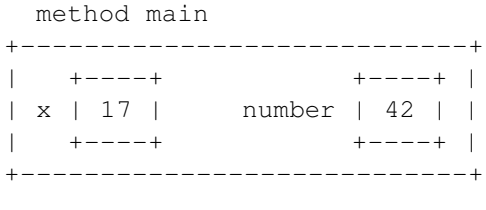
And we once again call doubleNumber, this time passing it the value of number. This is an odd situation because the parameter has the same name as the variable in main. But Java doesn't care that they have the same name. It always creates a new local variable for method doubleNumber:



So at this point in time, there are two different variables called number, one in each method. We then execute the statements of doubleNumber. We first report the value of number (42). Then we double it:



Again, notice that doubling number inside of doubleNumber has no effect on the original variable number that is in method main. These are separate variables. The method then reports the new value of number (84) and returns to main.



The program then reports the value of number and terminates. So the overall output for the program is as follows.

```
Initial value of number = 17
Final value of number = 34
x = 17

Initial value of number = 42
Final value of number = 84
number = 42
```

Thus, the local manipulations of the parameter do not change these variables outside the method. The fact that variables are copied is an important aspect of parameters. On the positive side, we know that variables are protected from change because the parameter will be a copy of the original. On the negative side, it means that although parameters will allow us to send values into a method, they will not allow us to get values back out of a method.

Multiple Parameters

So far the discussion of parameter syntax has been informal. It's about time that we wrote down more precisely what syntax we are using to declare static methods with parameters.

```
public static void <name>(<type> <name>, ..., <type> <name>) {
    <statement or variable declaration>;
    <statement or variable declaration>;
    ...
    <statement or variable declaration>;
}
```

This template indicates that we can declare as many parameters as we want inside the parentheses that appear after the name of a method in its header. We use commas to separate different parameters.

As an example of a method with multiple parameters, let's consider a variation of writeSpaces. It is convenient that we can tell it a different number of spaces to write, but it always writes spaces. What if we want 18 asterisks or 23 periods or 17 question marks? We can generalize the task even further by having the method take two parameters: both a character and a number of times to write that character.

```
public static void writeChars(char ch, int number) {
    for (int i = 1; i <= number; i++) {
        System.out.print(ch);
    }
}
```

The character to be printed is a parameter of type char, which we will discuss in more detail in the next chapter. Recall that character literals are enclosed in single quotation marks.

The syntax template for calling a method that accepts parameters is the following:

```
<method name>(<expression>, <expression>, ..., <expression>);
```

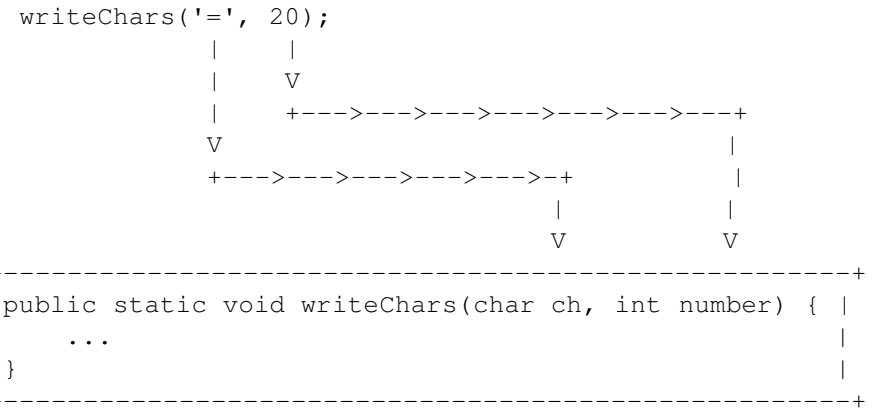
By calling this method we can write code like the following:

```
writeChars('=', 20);
System.out.println();
for(int i = 1; i <= 10; i++) {
    writeChars('>', i);
    writeChars(' ', 20 - 2 * i);
    writeChars('<', i);
    System.out.println();
}
writeChars('=', 20);
System.out.println();
```

which produces the following output:

```
====================
>                  <
>>                <<
>>>              <<<
>>>>            <<<<
>>>>>          <<<<<
>>>>>>        <<<<<<
>>>>>>>      <<<<<<<
>>>>>>>>    <<<<<<<<
>>>>>>>>>  <<<<<<<<<
>>>>>>>>>><<<<<<<<<<
====================
```

You can include as many parameters as you want when you define a method. Each method call must provide exactly that number of parameters. They are lined up sequentially. For example, consider the first call on writeChars in the code fragment above versus the header for writeChars. Java lines these two up in sequential order (the first actual parameter going into the first formal parameter, the second actual parameter going into the second formal parameter):



When writing methods that accept many parameters, the method header line can become very long. It is common to wrap long lines (ones that exceed roughly 80 characters in length) by inserting a line break after an operator or parameter and indenting the following line by twice the normal indentation width. For example:

```
// This method's header line is too long, so we'll wrap it.
public static void printTriangle(int xCoord1, int yCoord1,
        int xCoord2, int yCoord2, int xCoord3, int yCoord3) {
    ...
}
```

Parameters Versus Constants

How does this new technique relate to what you already know? We saw in Chapter 2 that class constants are a useful technique to increase the flexibility of our programs. By using such constants, you can make it easy to modify a program to behave differently. The parameter provides much of the same flexibility, and more. Consider the writeSpaces method. Suppose you wrote it using a class constant:

```
public static final int NUMBER_OF_SPACES = 10;
```

This gives you the flexibility to produce a different number of spaces, but has one major limitation. The constant can change only from execution to execution. It cannot change within a single execution. In other words, you can execute the program once with one value, edit the program, recompile and then execute it again with a different value. But you can't use different values in a single execution of the program using a class constant.

Parameters are more flexible. Because you specify the value to be used each time you call the method, you can use several different values in a single program execution. As you have seen, you can call the method many different times within a single program execution and have it behave differently every time. At the same time, however, the parameter is more work for the programmer than the class constant. It makes your method headers and method calls more tedious, not to mention making the execution (and, thus, the debugging) more complex.

Therefore, you will probably find occasion to use each technique. The basic rule is to use a class constant when you only want to change the value from execution to execution. If you want to use different values within a single execution, use the parameter.

Overloading of Methods

It is often the case that we want to create slight variations of the same method with different parameter passing. For example, we might have a method drawBox that allows you to specify a particular height and width but you might want to also have a version that draws a box of default size. In other words, sometimes you want to specify these values:

```
drawBox(8, 10);
```

and other times you want to just tell it to draw a box with the standard height and width:

```
drawBox();
```

Some programming languages require you to come up with different names for these. One might be called drawBox and the other might be called drawDefaultBox. Coming up with new names for each variation becomes tedious. Fortunately Java allows us to have more than one method with the same name as long as they have different parameters. This process is called *overloading* and the primary requirement for overloading is that the different methods that you define have different method signatures.

Method Signature

The name of a method along with its number and type of parameters.

Method Overloading

The ability to define two or more different methods with the same name as long as they have different method signatures.

For the drawBox example the two versions would clearly have different method signatures because one has two parameters and the other has zero parameters. It would be obvious from any call on the method which one to use. If you see two parameters, you execute the version with two parameters. If you see zero parameters, you execute the version with zero parameters.

It gets more complicated when overloading involves the same number of parameters, but this turns out to be one of the most useful applications of overloading. For example, the println method is actually a series of overloaded methods. We can call println passing it a String, or passing it an int, or passing it a double and so on. This flexibility is implemented as a series of different methods all of which take one parameter but one version takes a String, another version takes an int, another version takes a double and so on. Obviously you do slightly different things to print one of these kinds of data versus another, which is why it's useful to have these different versions of the method.

3.2 Methods that Return Values

We have been looking at action-oriented methods that perform some specific task. You can think of them as being like commands that you could give someone as in, "Draw a box" or "Draw a triangle." Parameters allow these commands to be more flexibile, as in, "Draw a box that is 10 by 20."

We will also want to be able to write methods that compute values. These methods are more like questions, as in "What is the square root of 2.5?" or "What do you get when you carry 2.3 to the 4th power?" For example, we can imagine having a method called "sqrt" that would compute the square root of a number.

It might seem that the way to write such a "sqrt" method would be to have it accept a parameter of type double and would println its square root to the console. But often we want to use the square root as part of a larger expression or computation, such as solving a quadratic equation or computing distance between points on an x/y plane.

A better solution would be a square root command where the number of interest is passed as a parameter, and its square root comes back to our program as a result. We could use the result as part of an expression, store it into a variable, or print it on the console. Such a command is a new type of method that is said to *return* a value.

Return

To send a value out as the result of a method, which can be used in an expression in your program. Void methods do not return any value.

If we had such a method, we could ask for the square root of 2.5 by writing code like this:

```
// assuming that we had a method named sqrt
double answer = sqrt(2.5);
```

The sqrt method has a parameter (the number to find the square root of), but it also returns a value (the square root). The actual parameter 2.5 goes "into" the method and the square root comes out. In the code above, we store the returned result in a variable called answer.

You can tell whether or not a method returns a value by looking at its header. We have been writing methods that all begin with `public static void`, as in:

```
public static void drawTriangle(int height)
```

The word `void` is known as the *return type* of the method:

```
public static void drawTriangle(int height)
              ~~~~
           return type
```

The void return type is a little odd because, as the word implies, it means that the method returns nothing. Instead of `void`, we can use any legal type. So we can write methods that return an int or that return a double or any other type. In the case of the method to compute square roots, we want it to return a double, so we would write its header as follows:

```
public static double sqrt(double n)
```

As in the previous case, the word that comes after `public static` is the return type of the method:

```
public static double sqrt(double n)
              ~~~~~~
            return type
```

Fortunately we don't need to write a method for computing the square root of a number because Java has one that is built-in. The method is included in a class known as Math that includes many useful computing methods. So before we discuss the details of writing our own methods that return values, let's explore the Math class and what it has to offer.

The Math Class

In Chapter 1 we mentioned that a great deal of predefined code has been written for Java that is collectively known as the *Java class libraries*. One of the most useful classes is called Math. It includes predefined mathematical constants and a large number of common mathematical functions. The Math class should be available on any machine that has Java properly installed.

As noted above, the Math class has a method called sqrt that computes the square root of a number. The method has the following header:

```
public static double sqrt(double n)
```

This header says that the method is called "sqrt", that it takes a parameter of type double and that it returns a value of type double.

Unfortunately, we can't just call this method directly by referring to it as "sqrt" because it is in another class. Whenever you want to refer to something declared in another class, you use the *dot notation*:

```
<class name>.<element>
```

So we refer to this method as Math.sqrt. So we might write a program like the following:

```
1  public class WriteRoots {
2      public static void main(String[] args) {
3          for (int i = 1; i <= 20; i++)
4              System.out.println("sqrt(" + i + ") = " + Math.sqrt(i));
5      }
6  }
```

which produces the following output:

```
sqrt(1) = 1.0
sqrt(2) = 1.4142135623730951
sqrt(3) = 1.7320508075688772
sqrt(4) = 2.0
sqrt(5) = 2.23606797749979
sqrt(6) = 2.449489742783178
sqrt(7) = 2.6457513110645907
sqrt(8) = 2.8284271247461903
sqrt(9) = 3.0
sqrt(10) = 3.1622776601683795
sqrt(11) = 3.3166247903554
sqrt(12) = 3.4641016151377544
sqrt(13) = 3.605551275463989
sqrt(14) = 3.7416573867739413
sqrt(15) = 3.872983346207417
sqrt(16) = 4.0
sqrt(17) = 4.123105625617661
sqrt(18) = 4.242640687119285
sqrt(19) = 4.358898943540674
sqrt(20) = 4.47213595499958
```

Notice that we are passing a value of type int to Math.sqrt, but the header says that it expects a value of type double. Remember that if Java is expecting a double and gets an int, it converts the int into a corresponding double.

The Math class also defines two constants that are frequently used, e and pi. Following the Java convention, we use all uppercase letters for their names and refer to them as Math.E and Math.PI.

Math Constants

Constant	Description
`E`	base used in natural logarithms (2.71828...)
`PI`	ratio of circumference of a circle to its diameter (3.14159...)

The following is a short list of some of the most useful static methods from the Math class.

Useful Static Methods in the Math Class

Method	Description	Example
`abs`	absolute value	`Math.abs(-308)` returns 308
`ceil`	ceiling (rounds upward)	`Math.ceil(2.13)` returns 3.0
`cos`	cosine (radians)	`Math.cos(Math.PI)` returns -1.0
`exp`	exponent base e	`Math.exp(1)` returns 2.7182818284590455
`floor`	floor (rounds downward)	`Math.floor(2.93)` returns 2.0
`log`	logarithm base e	`Math.log(Math.E)` returns 1.0
`log10`	logarithm base 10	`Math.log10(1000)` returns 3.0
`max`	maximum of two values	`Math.max(45, 207)` returns 207
`min`	minimum of two values	`Math.min(3.8, 2.75)` returns 2.75
`pow`	power (general exponentiation)	`Math.pow(3, 4)` returns 81.0
`random`	random value	`Math.random()` returns a random double value k such that $0.0 \leq k < 1.0$
`sin`	sine	`Math.sin(0)` returns 0.0
`sqrt`	square root	`Math.sqrt(2)` returns 1.4142135623730951
`toDegrees`	converts radian angles to degrees	`Math.toDegrees(Math.PI)` returns 180.0
`toRadians`	converts degree angles to radians	`Math.toRadians(270.0)` returns 4.71238898038469

You can see a complete list of methods defined in the Math class by checking out the API documentation for your version of Java. The acronym "API" stands for "Application Program Interface" or "Application Programming Interface." The API describes how to make use of the standard libraries that are available to Java programmers. It can be a bit overwhelming to read through the API documentation because the Java libraries are vast. So wander around a bit if you are so inclined, but don't be dismayed that there are so many libraries to choose from in Java.

If you do look into the Math API, you'll notice that the Math class has several overloaded methods. For example, there is a version of the absolute value method (Math.abs) for ints and another for doubles. The rules that govern which method is called are complex, so we won't cover them here. The basic idea, though, is that Java tries to find the method that is the best fit. For the most part, you don't have to think much about this issue. You can just let Java choose for you and it will generally make the right choice.

Defining Methods that Return Values

We can write our own methods that return values by using a special statement known as a *return* statement. For example, here is a method that takes a distance specified as a number of feet that returns the corresponding number of miles.

```
public static double miles(double feet) {
    return feet / 5280.0;
}
```

The miles method could be used by the main method in code such as the following.

```
System.out.println("15000 feet is " + miles(15000) + " miles.");
```

Notice once again that in the header for the method the familiar word `void` has been replaced with the word `double`. Remember that when you declare a method that returns a value, you have to tell Java what kind of value it will return. In fact, the keyword `void` simply means "no return value". We can update our syntax template for static methods once more to include the fact that the header includes a return type (void for none):

```
public static <type> <name>(<type> <name>, ..., <type> <name>) {
    <statement or variable declaration>;
    <statement or variable declaration>;
    ...
    <statement or variable declaration>;
}
```

The syntax of the return statement is:

```
return <expression>;
```

When Java encounters a return statement, it evaluates the given expression and immediately terminates the method, returning the value it obtained from the expression. Because of this, it's not legal to have any other statements after a return statement; the return must be the last statement in your method. It is also an error for a Java method with a non-void return type to terminate without a return.

There are exceptions to the previous rules that we'll see later. For example, it is possible that a method may have more than one return statement, although this won't come up until the next chapter, when we can do something called conditional execution using if and if/else statements.

Common Programming Error: Ignoring Return Value

When you call a method that returns a value, the expectation is that you'll do something with the value being returned. You can print it, store it into a variable, or use it as part of a larger expression. It is legal (but unwise) to simply call the method and ignore the value being returned from it:

```
miles(15000);   // doesn't do anything
```

But the preceding call doesn't print the number of miles or have any noticeable effect. If you want the value printed, you must do so yourself; simply calling the miles method never causes a println statement to execute.

```
double vacation = miles(15000);   // better
System.out.println("I drove " + vacation + " miles.");
```

A shorter form of the fixed code would be the following:

```
System.out.println("I drove " + miles(15000) + " miles.");
```

Let's look at another example method that returns a value. The Pythagorean Theorem of right triangles (as stated by the scarecrow in *The Wizard of Oz*) says that the length of the hypotenuse of a right triangle is equal to the square root of the sums of the squares of the two remaining sides. If you know the lengths of two sides *a* and *b* of a right triangle and want to find the length of the third side *c*, you compute it as follows:

$$c = \sqrt{a^2 + b^2}$$

So imagine that we want to print out the lengths of the hypotenuses of two right triangles: one with side lengths of 5 and 12, and the other with side lengths of 3 and 4. We could write code such as the following:

```
double c1 = Math.sqrt(Math.pow(5, 2) + Math.pow(12, 2));
System.out.println("hypotenuse 1 = " + c1);
double c2 = Math.sqrt(Math.pow(3, 2) + Math.pow(4, 2));
System.out.println("hypotenuse 2 = " + c2);
```

The preceding code is correct, but it's a bit hard to read, and we'd have to duplicate the same complex math a third time if we had a third triangle of interest. A better solution would be to create a method that computes and returns the hypotenuse length when given the two other side lengths as parameters. Such a method would look like this:

```
public static double hypotenuse(double a, double b) {
    double c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
    return c;
}
```

Our new hypotenuse method can be used to craft a more concise and readable main method. The overall program looks like the following:

```
 1  public class Triangles {
 2      public static void main(String[] args) {
 3          System.out.println("hypotenuse 1 = " + hypotenuse(5, 12));
 4          System.out.println("hypotenuse 2 = " + hypotenuse(3, 4));
 5      }
 6
 7      public static double hypotenuse(double a, double b) {
 8          double c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
 9          return c;
10          System.out.println(c);
11      }
12  }
```

A few variations of our program are possible. For one, it isn't necessary to store our return value into the variable c. We can simply compute and return the value in one line if we prefer. The body of the hypotenuse method would become the following:

```
return Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
```

Also, some programmers avoid using Math.pow for low powers such as 2 and just manually do the multiplication. If we'd done that here, the body of our hypotenuse method would have looked like this:

```
return Math.sqrt(a * a + b * b);
```

Common Programming Error: Statement After Return

It isn't legal to have other statements immediately following a return statement, because the statements can never be reached or executed. One common case where new programmers accidentally do this is when trying to print the value of a variable after returning. Imagine that we've written the hypotenuse method but have accidentally written the parameters to Math.pow in the wrong order, so our method is not producing the right answer. We try to debug this by printing the value of c that is being returned. Here's our faulty code:

```
// Trying to find the bug in this buggy version of hypotenuse.
public static double hypotenuse(double a, double b) {
    double c = Math.sqrt(Math.pow(2, a) + Math.pow(2, b));
    return c;
    System.out.println(c);   // this doesn't work
}
```

The compiler complains about the println statement being unreachable, since it follows a return statement. The compiler error output looks something like this:

```
Triangles.java:10: unreachable statement
        System.out.println(c);
        ^
Triangles.java:11: missing return statement
    }
    ^
2 errors
```

The fix is to move such a println statement earlier in the method, before the return statement.

```
// Trying to find the bug in this buggy version of hypotenuse.
public static double hypotenuse(double a, double b) {
    double c = Math.sqrt(Math.pow(2, a) + Math.pow(2, b));
    System.out.println(c);   // better
    return c;
}
```

3.3 Using Objects

We've spent a considerable amount of time discussing the primitive types in Java and how they work, so it's about time that we started talking about objects and how they work.

The idea for objects came from the observation that as we start working with a new kind of data (integers, reals, characters, text, etc), we find ourselves writing a lot of methods that operate on that data. It seemed odd to have these two things separated. It makes more sense to include some of the basic operations with the data itself. This packaging of data and operations into one entity is the central idea behind objects. An object stores some data and has methods that act on its data.

Object

A programming entity that contains state (data) and behavior (methods).

As we said in Chapter 1, classes are the basic building blocks of Java programs. But classes also serve another purpose: to describe new types of objects.

Class

A category or type of object.

When used this way, a class is like a blueprint of what the object looks like. Once we've given Java the blueprint, we can ask it to create actual objects that match the blueprint. We sometimes refer to each of the individual objects as *instances* of the class. We tend to use the words "instance" and "object" interchangeably.

This concept is difficult to understand in the abstract, so let's look at several different classes to better understand what it means and how it works. In keeping with our idea of focusing on fundamental concepts first, we're going to study how to use existing objects that are already part of Java, but we aren't going to study how to define our own new types of objects just yet. We'll get to that in Chapter 8 after we've had time to practice using objects.

Unfortunately, using objects requires some new syntax and concepts that differ from the primitive types we've seen. It would be nice if Java had a consistent model for using all types of data, but it doesn't. That means that we, unfortunately, have to learn two sets of rules if we want to understand how our programs operate: one for primitives and one for objects.

String Objects

Strings are one of the most useful and the most commonly used types of objects in Java, so we definitely want to see how they work. They don't make the best example of objects, though, because there are a lot of special rules that apply only to strings, so in the next section we'll look at a more typical kind of object.

Strings have the special property that there are literals that represent String objects. We've been using them in println statements since Chapter 1. What we haven't discussed is that these literal values represent objects of type String (instances of the String class). For example, in the same way that you can say:

```
int x = 8;
```

you can say:

```
String s = "hello there";
```

We can declare variables of type String and can use the assignment statement to give a value to these variables. We can also write code that involves String expressions:

```
String s1 = "hello";
String s2 = "there";
String combined = s1 + " " + s2;
```

This code defines two Strings that each represent a single word and a third String that represents the concatenation of the two words with a space in between. You'll notice that the type `String` is capitalized (as are the names of all object types in Java), unlike the primitive types such as double and int.

So far we haven't seen anything special about String objects. Remember that the idea behind objects was to include basic operations with the data itself, the way we build a car that has controls built in. The data stored in a String is a sequence of characters. There are all sorts of operations we might want to perform on this sequence of characters. For example, we might want to know how many characters there are in the String. String objects have a length method that returns this information.

If the length method were static, you would call it by saying something like:

```
length(s)     // this isn't legal
```

But when performing operations on objects, we use a different syntax. Objects store data and methods, so the method to report a String's length actually exists inside that String object itself. To call an object's method, you write the name of the variable first and then the name of the method with a dot in between:

```
s.length()
```

Think of it as talking to the String object. When you ask for `s.length()`, you're saying, "Hey, s. I'm talking to you. What's your length?". Of course, different String objects have different lengths, so you will get different answers when you talk to different String objects.

The general syntax for calling a method of an object is the following:

```
<object's name> . <method name> ( <parameter(s)> )
```

For example, suppose that we have initialized two string variables as follows:

```
String s1 = "hello";
String s2 = "how are you?";
```

We can use a println to examine the length of each string:

```
System.out.println("Length of s1 = " + s1.length());
System.out.println("Length of s2 = " + s2.length());
```

which produces the following output:

```
Length of s1 = 5
Length of s2 = 12
```

What else might we want to do with a String object? With the length method we can figure out how many characters there are, but what about getting the individual characters themselves? There are several ways to do this, but one of the most common is to use a method called charAt that returns the character at a specific location in the String.

This leads us to the problem of how to specify locations in a sequence. Obviously there is a first character, second character, and so on, so it makes sense to use an integer to refer to a specific location. We call this the *index* and, as we'll see, Java usually uses 0 as the first index value.

Index

An integer used to specify a location in a sequence of values. Java generally uses 0-based indexing (0 as the first index value followed by 1, 2, 3 and so on).

So each character of a String object is assigned an index starting with index 0. For example, for our variable s1 that refers to the String "hello" the indexes would be:

```
h    e    l    l    o
|    |    |    |    |
0    1    2    3    4
```

It may seem intuitive to consider the letter "h" to be at position 1, but there are advantages to starting with an index of 0 and it's a convention that was adopted by the designers of the C language that has been followed also by the designers of C++ and Java, so it's a convention you'll have to learn to live with. For our longer string s2 the positions would be:

```
h    o    w         a    r    e         y    o    u    ?
|    |    |    |    |    |    |    |    |    |    |    |
0    1    2    3    4    5    6    7    8    9   10   11
```

Notice that the spaces in the string have positions as well, as in positions 3 and 7 above. Also notice that the indexes for a given String always range from 0 to one less than the length of the String.

Using the charAt method, we can request specific characters of a String. The return type is char. For example, if we ask for s1.charAt(1) we'll get 'e' (the 'e' in "hello"). If we ask for s2.charAt(5) we'll get 'r' (the 'r' in "how are you?"). For any String, if we ask for charAt(0), we'll get the first character of the String.

When working with String objects, we often find it useful to write a for loop to handle the different characters of the String. Because Strings are indexed starting at 0, this task is easier to write with for loops that start with 0 rather than 1. Consider, for example, the following code that prints out the individual characters of s1:

```
String s1 = "hello";
for (int i = 0; i < s1.length(); i++) {
    System.out.println(i + ": " + s1.charAt(i));
}
```

which produces the following output:

```
0: h
1: e
2: l
3: l
4: o
```

Remember that when we start loops at 0, we usually test with less-than ("<") rather than less-than-or-equal ("<="). Our String s1 has 5 characters in it, so the call on s1.length() will return 5. But because the first index is 0, the last index will be one less than 5 (4). This convention takes a while to get used to, but 0-based indexing is used throughout Java, so you'll eventually get the hang of it.

Another useful String method is the substring method. It takes two integer arguments representing a starting and ending index. When you call the substring method, you provide two of these indexes, the index of the first character you want and the index just past the last index that you want.

Recall that our String s2 that we set to "how are you?" has the following positions:

```
h    o    w         a    r    e         y    o    u    ?
|    |    |    |    |    |    |    |    |    |    |    |
0    1    2    3    4    5    6    7    8    9   10   11
```

If you want to pull out the individual word "how" from this String, you'd ask for:

```
s2.substring(0, 3)
```

Remember that the second value that you pass to the substring method is supposed to be one beyond the end of the substring you are forming. So even though there is a space at position 3 in the original string, it will not be part of what we get from the call on substring. Instead we get all characters just before position 3.

Following this rule means that sometimes you will give a position to substring at which there isn't a character. The last character in the string that s2 refers to is at index 11 (the question mark). If you want to get the substring "you?" including the question mark, you'd ask for:

```
s2.substring(8, 12)
```

There is no character at position 12 in s2, but this call asks for characters starting at position 8 that come before position 12, so this actually makes sense.

You have to be careful about what indexes you use. With the substring method we can ask for the position just beyond the end of the String, but you can't ask for anything beyond that. For example, if you ask for:

```
s2.substring(8, 13)   // out of bounds!
```

Your program will generate an execution error. Similarly, if you ask for the charAt a nonexistent position, your program will generate an execution error. These errors are known as *exceptions*.

Exceptions are runtime errors as mentioned in Chapter 1.

Exception

A runtime error that prevents a program from continuing its normal execution.

We say that an exception is *thrown* when an error is encountered. When an exception is thrown, Java looks to see if you have written code to handle it. If not, program execution is halted and you will see what is known as a *stack trace* or *back trace*. The stack trace shows you the series of methods that have been called in reverse order. In this case of bad String indexes, the exception prints a message such as the following to the console:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of r
    at java.lang.String.substring(Unknown Source)
    at ExampleProgram.main(ExampleProgram.java:3)
```

You can use Strings as parameters to methods. For example, the following program uses String parameters to eliminate some of the redundancy in a popular children's song.

```
 1  public class BusSong {
 2      public static void main(String[] args) {
 3          verse("wheels", "go", "round and round");
 4          verse("wipers", "go", "swish, swish, swish");
 5          verse("horn", "goes", "beep, beep, beep");
 6      }
 7
 8      public static void verse(String item, String verb, String sound) {
 9          System.out.print("The " + item + " on the bus " + verb + " ");
10          System.out.println(sound + ",");
11          System.out.println(sound + ",");
12          System.out.println(sound + ".");
13          System.out.print("The " + item + " on the bus " + verb + " ");
14          System.out.println(sound + ",");
15          System.out.println("All through the town.");
16          System.out.println();
17      }
18  }
```

It produces the following output:

```
The wheels on the bus go round and round,
round and round,
round and round.
The wheels on the bus go round and round,
All through the town.

The wipers on the bus go swish, swish, swish,
swish, swish, swish,
swish, swish, swish.
The wipers on the bus go swish, swish, swish,
All through the town.

The horn on the bus goes beep, beep, beep,
beep, beep, beep,
beep, beep, beep.
The horn on the bus goes beep, beep, beep,
All through the town.
```

The following are some of the most useful methods that you can call on String objects.

Useful Methods of String Objects

Method	Description	Example (assuming s is "hello")
`charAt(index)`	character at a specific index	`s.charAt(1)` returns `'e'`
`endsWith(text)`	whether or not string ends with some text	`s.endsWith("llo")` returns `true`
`indexOf(text)`	index of a particular character or String (-1 if not present)	`s.indexOf("o")` returns `4`
`length()`	number of characters	`s.length()` returns `5`
`startsWith(text)`	whether or not string starts with some text	`s.startsWith("hi")` returns `false`
`substring(start, stop)`	characters from start index to just before stop index	`s.substring(1, 3)` returns `"el"`
`toLowerCase()`	a new string with all lowercase letters	`s.toLowerCase()` returns `"hello"`
`toUpperCase()`	a new string with all uppercase letters	`s.toUpperCase()` returns `"HELLO"`

Strings in Java are immutable, which means that once they are constructed, they can never be changed in value.

Immutable Object

An object whose value cannot be changed.

It may seem odd that strings are immutable and yet they have methods like toUpperCase and toLowerCase. You have to read the table description carefully. These methods don't actually change a given string object, they return a new string. Consider the following code.

```
String s = "Hello Maria";
s.toUpperCase();
System.out.println(s);
```

You might think that this will turn the string to its uppercase equivalent, but it doesn't. The second line of code constructs a new string that has the uppercase equivalent of the string, but we don't do anything with this new value. The key is to either store this new string in a different variable or to reassign the variable s to point to the new string:

```
String s = "Hello Maria";
s = s.toUpperCase();
System.out.println(s);
```

This version of the code produces the following output:

```
HELLO MARIA
```

The toUpperCase and toLowerCase methods are particularly helpful when you want to perform string comparisons in which you ignore the case of the letters involved.

Point Objects

Strings are extremely useful objects, but the rules for creating and using them don't completely match those of other types of objects. We'll now examine a more typical type of object: Point. A Point object stores the (x, y) coordinates of a point in 2-D space. These coordinates are expressed as integers, although there are also variations for storing points expressed using floating point numbers.

Strings are a special case that have literal values that can be referred to directly. Most objects have to be explicitly constructed by calling a special method known as a constructor.

Construct (Constructor)

A method that creates and initializes an object. Objects in Java programs must be constructed before they can be used.

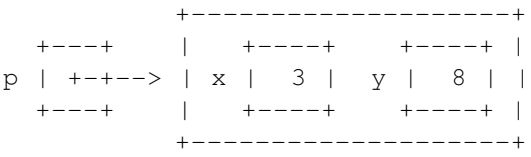
Remember that a class is like a blueprint for a family of objects. Calling a constructor is like sending an order to the factory asking it to follow the blueprint to get you an actual object that you can manipulate. When you send in your order to the factory, you sometimes specify certain parameters (e.g., what color you want the object to be).

In Java, constructors are called using the special keyword `new`, followed by the object's type and any necessary parameters. For example, to construct a specific Point object, you have to pass the values you want for x and y:

```
Point p = new Point(3, 8);
```

This code performs a call on a constructor method of the Point class. Constructors always have the same name as the class. In this case, we pass the constructor two integer values as parameters to specify the x-coordinate and y-coordinate of the point.

After executing the line of code above, we would have the following situation:



Once you have constructed a Point object, what can you do with it? One of the most common things you do with an object is to print it to the console. A Point object, like many Java objects, can be printed with the println statement.

```
System.out.println(p);
```

The println statement produces the following output:

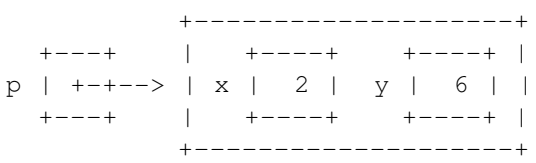
```
java.awt.Point[x=3,y=8]
```

The output produced begins with the full name of the class (java.awt.Point). The "java.awt" part of this name is explained in a moment. After the name of the class, we have the x and y coordinates listed inside square brackets. This format is a little ugly, but it lets you see the x and y values inside a given point.

Point objects also have a method called "translate" that can be used to shift the coordinates by a specific delta-x and delta-y, which are passed as parameters. When you translate a point, you shift its location by the specified amount. For example, you might say:

```
p.translate(-1, -2);   // subtract 1 from x, subtract 2 from y
```

Given that the point started out with coordinates (3, 8), this translation would leave the point with coordinates (2, 6). Thus, after this line of code is executed, we end up with the following situation:



Here is a complete program that constructs a Point object and translates its coordinates, using println statements to examine the coordinates before and after the call.

```
 1  import java.awt.*;
 2
 3  public class PointExample1 {
 4      public static void main(String[] args) {
 5          Point p = new Point(3, 8);
 6          System.out.println("initially p = " + p);
 7          p.translate(-1, -2);
 8          System.out.println("after translating p = " + p);
 9      }
10  }
```

This code produces the following output:

```
initially p = java.awt.Point[x=3,y=8]
after translating p = java.awt.Point[x=2,y=6]
```

There is something new at the beginning of this class file called an import declaration. Remember that Java has a large number of classes included in what are collectively known as the Java class libraries. To help manage these classes, Java provides an organizational unit known as a package. Related classes are combined together into a single package.

Package

A collection of related Java classes.

For example, the Point class is stored in a package known as java.awt, which is an abbreviation for the "Java Abstract Windowing Toolkit". Java programs don't normally have access to a package unless they include an import declaration.

Import Declaration

A request to access a specific Java package.

We haven't needed an import statement yet because Java automatically imports every class stored in a package called "java.lang". The java.lang package includes basic classes that most Java programs would be likely to use (e.g., System, String, Math). Because Java does not automatically import java.awt, we have to do it ourselves.

The import declaration allows you to import just a single class from a package, as in:

```
import java.awt.Point;
```

Some people prefer to specifically mention each class that you are importing. The problem is that once you start importing one class from a package, you're likely to want to import others as well. Java allows you to use an asterisk to import all classes from a package:

```
import java.awt.*;
```

We will use the asterisk version of import in this book to keep things simple.

Reference Semantics

Objects are stored in the computer's memory in a different way than primitive data. For example, when we declare an integer variable:

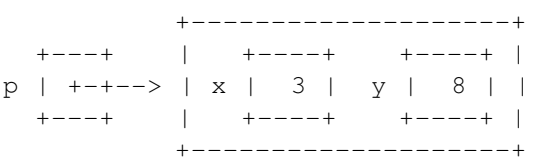
```
int x = 8;
```

The variable stores the actual data. So we've drawn pictures like the following:

The situation is different for objects. With objects, the variable doesn't store the actual data. Instead, the data is stored in an object and the variable stores a reference to where the object is stored. So we have two different elements in memory: the variable and the object. So when we construct a Point object:

```
Point p = new Point(3, 8);
```

we end up with the following:



As the diagram indicates, we have two different values stored in memory. We have the Point object itself, which appears on the right side of this picture. We also have a variable called p which stores a reference to the Point object (represented in this picture as an arrow). We say that p *refers* to the object.

This convention will take some time to get used to because these are two different approaches to storing data. These approaches are so common that computer scientists have technical terms to describe them. The system for the primitive types like int is known as value semantics and those types are often referred to as value types. The system for Points and other objects is known as reference semantics and those types are often referred to as reference types.

Value Semantics

A system in which values are stored directly and copying involves the creation of independent copies of values. Types that use value semantics are called value types.

Reference Semantics

A system in which references to values are stored and copying involves copying references. Types that use reference semantics are called reference types.

It will take us a while to explore all of the implications of this difference. The key thing to remember is that when you are working with objects, you are always working with references to data rather than the data itself.

At this point you are probably wondering why Java has two different systems. Java was designed for object-oriented programming, so the first question to consider is why Sun decided that objects should have reference semantics. There are two primary reasons:

- **efficiency:** Objects can be complex which means that they can take up a lot of space in memory. We don't want to have to make copies of such objects because we would run out of memory. For example, a String object might take up a lot of space in memory because it might store a large number of characters. But even if such a String object is very large, a reference to it can be fairly small in the same way that even a house that is a mansion will have a simple street address. As another analogy, think of how people use cell phones. The phones can be very tiny and easy to transport because cell phone numbers don't take up much space. Imagine how different it would be if instead of carrying around a set of cell phone numbers you tried to carry around the actual people instead.

- **sharing:** It is often the case that having a copy of something is not good enough. Suppose that your instructor tells all of the students in the class to put their tests into a certain box after the exam is over. Imagine how confusing it would be if each student made a copy of the box. The obvious intent is that all of the students will be using the same box. With reference semantics, you can have many references to a single object, which allows different parts of your program to share a certain object.

So without reference semantics, Java programs would be more difficult to write. Then why did Sun decide to include primitive types that have value semantics? The reasons are primarily historical. Sun wanted to leverage the popularity of C and C++, which had similar types, and Sun wanted to

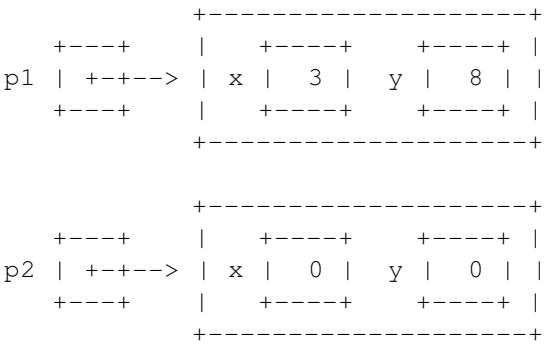
guarantee that Java programs would run quickly, which was easier to accomplish with the more traditional primitive types. If Sun had a chance to redesign Java today, they might very well get rid of the primitive types and go with a consistent object model with just reference semantics.

Multiple Objects

In the last section we saw how to manipulate a single object. Consider the following lines of code that construct two different objects.

```
Point p1 = new Point(3, 8);
Point p2 = new Point();
```

Each object must be constructed separately by a call on new. Remember that we call these *instances* of the class. The first call passes the x-coordinate and y-coordinate of the point. The second call uses a different constructor with a different method signature (zero parameters instead of two parameters). This constructor sets the x-coordinate and y-coordinate to 0. The 0-argument constructor is often referred to as the *default* constructor. After executing these statements, our memory would look as follows.

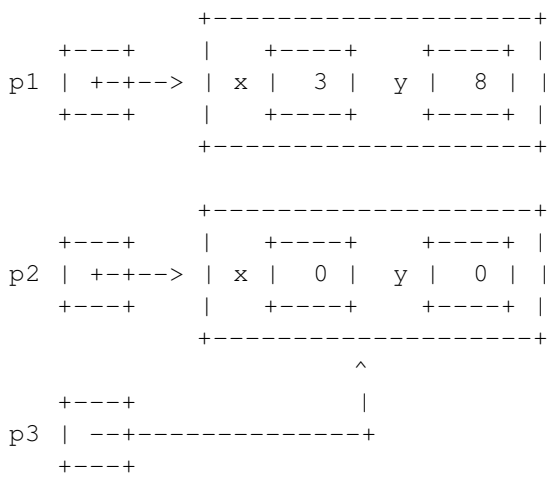


To underscore the fact that variables and objects are stored separately, consider what would happen if you now execute the following code:

```
Point p3 = p2;
```

This declares a third Point variable but doesn't include a third call on new.

That means that we still have just the two Point objects even though we now have three Point variables:



The variables p2 and p3 both refer to the same Point object. This situation doesn't arise when you use value types like int. It happens only with objects.

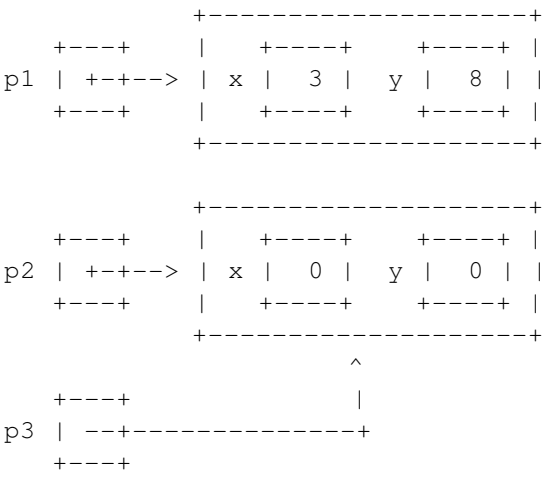
Let's look at a complete program to explore how this works.

```
 1  import java.awt.*;
 2
 3  public class PointExample2 {
 4      public static void main(String[] args) {
 5          // declare variables and construct objects
 6          Point p1 = new Point(3, 8);
 7          Point p2 = new Point();
 8          Point p3 = p2;
 9
10          // see what is in each point initially
11          System.out.println("p1 = " + p1);
12          System.out.println("p2 = " + p2);
13          System.out.println("p3 = " + p3);
14          System.out.println();
15
16          // three translations
17          p1.translate(-1, -2);
18          p2.translate(4, 8);
19          p3.translate(2, 3);
20
21          // see what is in each point now
22          System.out.println("p1 = " + p1);
23          System.out.println("p2 = " + p2);
24          System.out.println("p3 = " + p3);
25      }
26  }
```

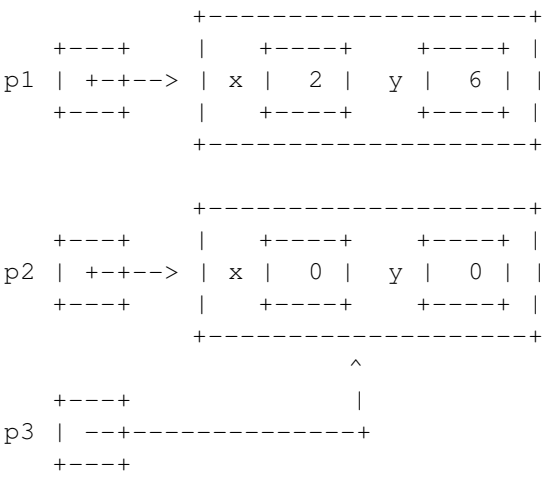
This program produces output that might be surprising:

```
p1 = java.awt.Point[x=3,y=8]
p2 = java.awt.Point[x=0,y=0]
p3 = java.awt.Point[x=0,y=0]

p1 = java.awt.Point[x=2,y=6]
p2 = java.awt.Point[x=6,y=11]
p3 = java.awt.Point[x=6,y=11]
```

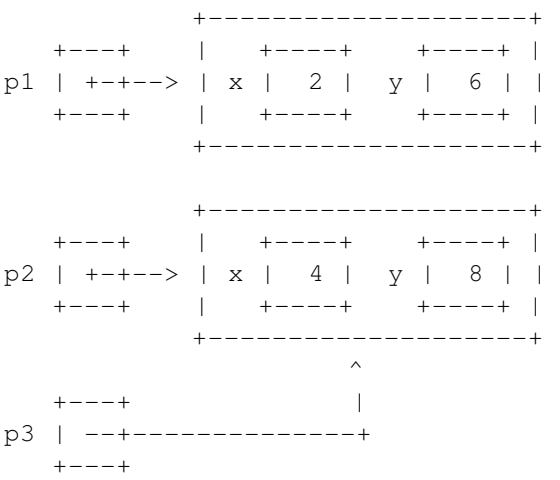
We end up with coordinates (6, 11), but there aren't any 6's or 11's in the original program. Let's see how this happens. The manipulations using variable p1 are fairly straightforward. Variables p2 and p3 are more complex. Remember that we have a single Point object that both of these variables refer to. This object initially has coordinates (0, 0). So after executing the first three lines of code, memory looks like this:



When we execute the first set of println statements we see that p1 has coordinates (3, 8). When we execute the println statement on p2 we get the coordinates (0, 0) and when we execute the println statement on p3 we get these coordinates again, because each variable is referring to this object. Then we translate the coordinates for p1 by -1 in the x direction and -2 in the y direction:

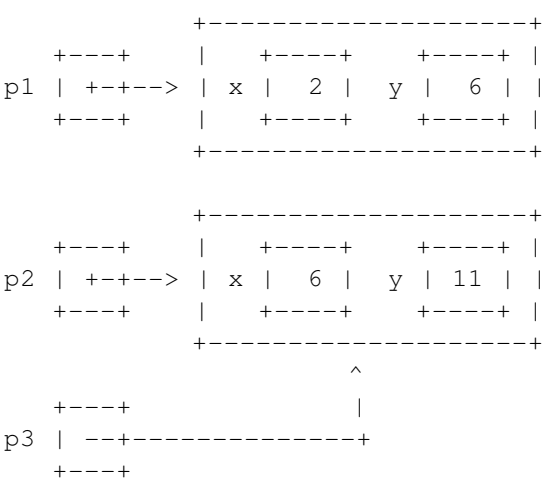


Then we translate the coordinates for p2 by 4 in the x direction and by 8 in the y direction:



Then we translate the coordinates for p3 by 2 in the x direction and 3 in the y direction.

But because p3 refers to the same object as p2, this translation is, in effect, added to the previous one:



We started with the coordinates (0, 0) and we translated p2 by (4, 8) and p3 by (2, 3), but p2 and p3 refer to the same object, which means we ended up translating that single point twice, leaving it at (6, 11). The final println statements report the new coordinates of p1 and report the new coordinates of the other point twice (once through the println on p2 and once through the println on p3).

Objects as Parameters to Methods

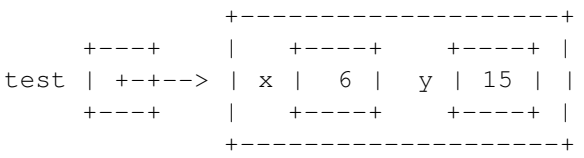
Consider what happens when you define a static method that uses a reference type for its parameter:

```
public static void manipulate(Point p) {
    p.translate(2, 3);
}
```

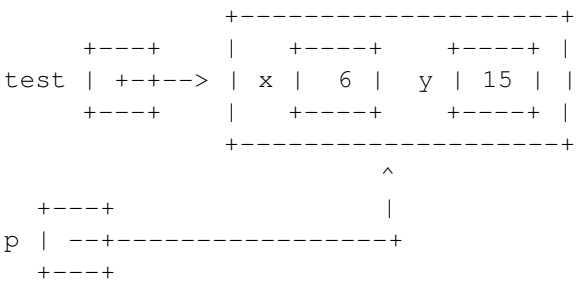
Remember that a parameter becomes a copy of whatever is passed in the call. With the value types like int, any changes that we made to the parameter have no effect on any variable passed to the method. The situation is more complicated for objects because of their reference semantics. Suppose that we have defined a Point variable and we pass it to this method:

```
Point test = new Point(6, 15);
manipulate(test);
```

Does the method end up translating the coordinates of the object? The answer is yes, even though we are using a parameter that creates a copy. Think of what is happening in the computer's memory. When we declare our variable test, we end up with this situation:



When we call the manipulate method, Java makes a copy of the variable test and stores this in the parameter called p. But the variable test isn't itself an object. It stores a reference to an object. So when we make a copy of it, we end up with this situation in memory:

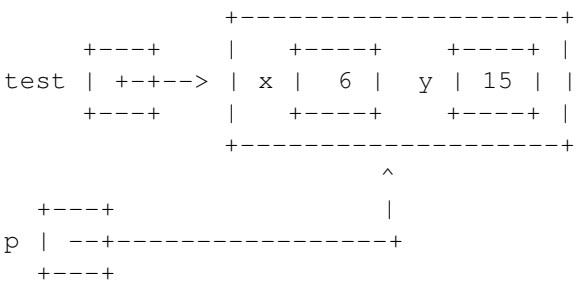


We now have two variables that refer to the same Point object. So it doesn't matter that p is a copy, because it's referring to the same object as the original. So any calls on translate using p will change the object that test refers to.

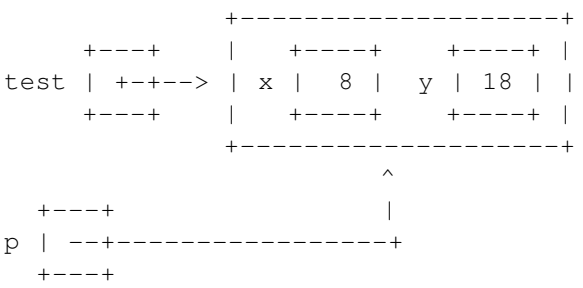
There is one other case worth considering. What if the manipulate method had been written this way instead:

```
public static void manipulate(Point p) {
    p.translate(2, 3);
    p = new Point(17, 45);
}
```

Let's look at this in detail. When the method is called, we set up the variable p as a copy of the variable test, which leads to two variables referring to the same Point object.

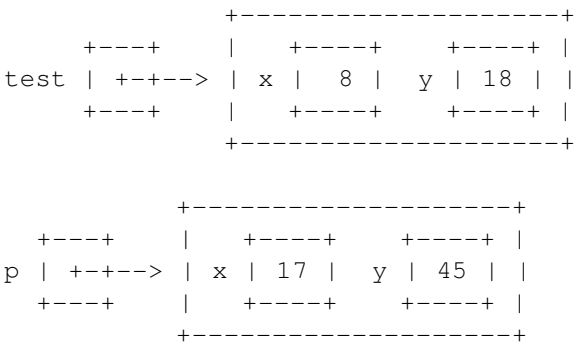


We translate the coordinates for p by 2 in the x direction and 3 in the y direction as in the old version of the method:



Then we turn around and reset the variable p to a new Point object with a different set of coordinates. Does this affect our variable test? The answer is no.

Because p is a local copy, changing its value has no effect on our variable test:



The bottom line is that when you pass an object as a parameter, you can change the object itself but you can't change the variable that points to the object.

Phone numbers provide a useful analogy. Suppose that you have a piece of paper on which you have written down a person's phone number. That gives you the abilty to call that person. You can make a copy of that piece of paper and give it to someone else, which is similar to what happens when you pass an object as a parameter to a method. Because you have your own piece of paper telling you the phone number, you will never lose it. But the other person has the phone number as well, which means they can make a phone call just as you can.

3.4 Interactive Programs

We have seen that by calling System.out.println and System.out.print we can produce output in the console window rather easily. We can also write programs that pause and wait for the user to type a response. Such programs are known as *interactive* programs and the responses typed by the user are known as *console input*.

Console Input

Responses typed by the user when an interactive program pauses for input.

When you refer to System.out, you are accessing an object in the System class known as the standard output stream, or "standard out" for short. There is a corresponding object for standard input known as System.in. But Java wasn't designed for console input and System.in has never been particularly easy to use for console input. Fortunately for us there is an easier way to read console input.

Scanner Objects

Starting with version 1.5, Java has a class called Scanner that simplifies reading from the console and reading from files. It is part of the java.util package, so we will have to remember to include the following import declaration in our programs that use Scanner:

```
import java.util.*;   // for Scanner
```

To use a Scanner you first have to construct one. You can construct a Scanner by passing it a reference to an input stream. To read from the console window, we pass it System.in:

```
Scanner console = new Scanner(System.in);
```

Once constructed, you can ask the Scanner to return a value of a particular type. There are a number of different methods that all begin with the word "next" to obtain these various types of values.

Scanner Methods

Method	Description
`next()`	reads and returns the next token as a String
`nextDouble()`	reads and returns a double value
`nextInt()`	reads and returns an int value
`nextLine()`	reads and returns the next line of input as a String

Typically we will use a variable to keep track of the value returned by one of these methods. For example, we might say:

```
int n = console.nextInt();
```

The call on the console object's nextInt method pauses for user input. Whenever the computer pauses for input, it will pause for an entire line of input. In other words, whenever the computer pauses for input, it will wait until the user hits the "Enter" key before continuing execution.

You can use the Scanner class to read input line by line using the nextLine method, although we won't be using nextLine very much for now. The other "next" methods are all token based.

Token

A single element of input (e.g., one word, one number).

By default, the Scanner uses whitespace to separate tokens.

Whitespace

Spaces, tab characters and "new line" characters.

A Scanner object looks at what the user types and uses the whitespace on the input line to break it up into individual tokens. For example, the following line of input:

```
hello      there. how are         "you?"  all-one-token
```

would be split into six tokens:

```
hello
there.
how
are
"you?"
all-one-token
```

Notice that the Scanner includes punctuation characters like periods, question marks, dashes and quotation marks in the tokens it generates. Also notice that we get just one token for "all-one-token". That's because there is no whitespace in the middle to break it up into different tokens. You can control how a Scanner turns things into tokens (a process called *tokenizing* the input), but we won't be doing anything that fancy.

It is possible to read more than one value from the Scanner, as in:

```
double x = console.nextDouble();
double y = console.nextDouble();
```

Because we have two different calls on the console object's nextDouble method, this code will cause the computer to pause until the user has entered two number values. The values can be entered on the same line or on separate lines. In general, the computer continues to pause for user input until it has obtained whatever values you have asked the Scanner to obtain.

If a user types something that isn't an integer when you're calling nextInt, such as XYZZY, the Scanner object generates an exception. Recall from the section on String objects that exceptions are runtime errors that halt program execution. In this case, a runtime error output such as the following appears.

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at Example.main(Example.java:13)
```

We will see in a later chapter how to test for user errors. In the meantime, we will assume that the user provides appropriate input.

A Sample Interactive Program

Using the Scanner class, we can write a complete interactive program that performs a useful computation for the user. If you ever find yourself buying a house, you'll want to know what your monthly payment for the loan is going to be. The following is a complete program that asks for information about a loan and prints the monthly payment.

```
 1  // This program prompts for information about a loan and computes
 2  // the monthly loan payment.
 3
 4  import java.util.*;   // for Scanner
 5
 6  public class Mortgage {
 7      public static void main(String[] args) {
 8          Scanner console = new Scanner(System.in);
 9
10          // obtain values
11          System.out.println("This program computes monthly loan payments.");
12          System.out.print("loan amount     : ");
13          double loan = console.nextDouble();
14          System.out.print("number of years : ");
15          int years = console.nextInt();
16          System.out.print("interest rate   : ");
17          double rate = console.nextDouble();
18          System.out.println();
19
20          // compute result and report
21          int n = 12 * years;
22          double c = rate / 12.0 / 100.0;
23          double payment = loan * c * Math.pow(1 + c, n)
24                           / (Math.pow(1 + c, n) - 1);
25          System.out.println("monthly payment = $" + (int) payment);
26      }
27  }
```

The following is a sample execution of the program:

```
This program computes monthly loan payments.
loan amount     : 275000
number of years : 30
interest rate   : 6.75

monthly payment = $1783
```

The first thing we do in the program is to construct a Scanner object that we use for console input. Then the program explains what is going to happen. This is essential for interactive programs. You don't want a program to pause for user input until you've explained to the user what is going to happen.

Then you'll notice several pairs of statements like these:

```
System.out.print("loan amount     : ");
double loan = console.nextDouble();
```

The first statement is called a *prompt*, a request for information from the user. We use a print statement instead of a println so that the user will type the values on the same line as the prompt (i.e., to the right of the prompt). The second statement calls the nextDouble method of the console object to read a value of type double from the user. This value is stored in a variable called loan. This pattern of prompt/read statements is common in interactive programs.

After prompting for values, the program computes several values. The formula for computing monthly mortgage payments involves the loan amount, the total number of months involved (a value we call "n") and the monthly interest rate (a value we call "c"). The payment formula is given by the following equation:

$$payment = loan \frac{c(1+c)^n}{(1+c)^n - 1}$$

You will notice in the program that we use the Math.pow method for exponentiation to translate this formula into a Java expression.

The final line of the program prints out the monthly payment. You might imagine that we would simply say:

```
System.out.println("monthly payment = $" + payment);
```

Because the payment is stored in a variable of type double, this would print all of the digits of the number. For example, for the log listed above, it would print the following:

```
monthly payment = $1783.6447655625927
```

That is a rather strange looking output for someone used to dollars and cents. For the purposes of this simple program, it's easy to cast this to an int and report just the dollar amount of the payment:

```
System.out.println("monthly payment = $" + (int) payment);
```

Most people trying to figure out a mortgage aren't that interested in the pennies, so the program is still useful. In the next section we will see how to round a number like this to two decimal places.

3.5 Case Study: Projectile Trajectory

It's time to pull together the different threads of this chapter with a more complex example that will involve parameters, methods that return values, mathematical computations and the use of a Scanner object for console input.

Physics students are often asked to consider the trajectory that a projectile will follow given its initial velocity and its initial angle relative to the horizontal. For example, the projectile might be a football that has been kicked by a person. We want to compute the path it follows given Earth's gravity. To keep the computation reasonable, we will ignore air resistance.

There are several questions that we generally want to answer for such a problem:

- When does the projectile reach its highest point?
- How high does it reach?

- How long does it take to come back to the ground?
- How far does it land from where it was launched?

You could imagine several ways to answer these questions. One simple approach is to provide a table that displays the trajectory step by step, indicating the x position, y position and elapsed time.

To make such a table, we need to obtain three values from the user: the initial velocity, the angle relative to the horizontal and the number of steps to include in the table we will produce. We could ask for the velocity either in meters/second or feet/second. Given that this is a Physics problem, we'll stick to the Metric System and ask for meters/second.

We also have to think about how to specify the angle. Unfortunately, most of the Java methods that operate on angles require an angle in radians rather than degrees. We could ask for the user to give us the angle in radians, but that would be highly inconvenient for the user. Instead, we can allow the user to enter the angle in degrees and we can convert it to radians using the built-in method Math.toRadians.

So the interactive part of our program would look like this:

```
Scanner console = new Scanner(System.in);
System.out.print("velocity (meters/second)? ");
double velocity = console.nextDouble();
System.out.print("angle relative to horizontal (degrees)? ");
double angle = Math.toRadians(console.nextDouble());
System.out.print("number of steps to display? ");
int steps = console.nextInt();
```

Notice that for the velocity and angle, we call the nextDouble method of the console object because we want to let the user specify any number (including one with a decimal point), but for the number of steps we call nextInt because the number of lines in our table needs to be an integer.

Look more closely at this line of code:

```
double angle = Math.toRadians(console.nextDouble());
```

Some beginners would write this as two separate steps:

```
double angleInDegrees = console.nextDouble();
double angle = Math.toRadians(angleInDegrees);
```

Both approaches work and both approaches are reasonable, but keep in mind that you don't need to break this up into two separate steps. You can write it in the more compact form as a single line of code.

Once we have obtained these values from the user, we are ready to begin the computations for our trajectory table. We need to compute the x component of the velocity versus the y component of the velocity. From Physics we know that these can be computed as follows:

```
double xVelocity = velocity * Math.cos(angle);
double yVelocity = velocity * Math.sin(angle);
```

Because we are ignoring the possibility of air resistance, the x-velocity will not change. The y-velocity is subject to the pull of gravity. Physics tells us that on the surface of the Earth, gravity is approximately 9.81 meters/second^2. This is an appropriate value to define as a class constant:

```
public static final double EARTH_ACCELERATION = -9.81;
```

Notice that we define gravity as a negative number because it decreases the y-velocity of an object (pulling it down as opposed to pushing it away).

Our goal is to display x, y and elapsed time as the object goes up and comes back down again. The y-velocity decreases steadily until it becomes 0. From Physics we know that there is a symmetry in this problem. The projectile will go upward until its y-velocity reaches 0 and then it will follow a similar path back down that takes an equal amount of time. Thus, the total time involved in seconds can be computed as follows:

```
double totalTime = -2.0 * yVelocity / EARTH_ACCELERATION;
```

Now how do we compute the values of x, y and elapsed time to include in our table? Two of these are relatively simple. We want steady time increments for each different entry in the table, so we can compute the time increment by dividing total time by the number of steps we want to include in our table:

```
double timeIncrement = totalTime / steps;
```

As noted earlier, the x-velocity does not change, so for each of these time increments, we move the same distance in the x direction:

```
double xIncrement = xVelocity * timeIncrement;
```

The tricky value to compute here is the y position. Because of gravity the y velocity changes over time. But from Physics we have the following general formula for computing the displacement of an object given the velocity v, time t and acceleration a:

$$\text{displacement} = vt + \tfrac{1}{2}at^2$$

In our case, the velocity we want is the y-velocity and the acceleration is Earth's gravity. Here, then, is a pseudocode description of how to create the table:

```
set all of x, y and t to 0
for (given number of steps) {
    t += timeIncrement;
    x += xIncrement;
    y = yVelocity * t + 0.5 * EARTH_ACCELERATION * t * t;
    report step #, x, y, t
}
```

We are fairly close to having real Java code here, but we have to think about how to report the values of x, y and t in a table. They will all be of type double. That means they are likely to produce a large number of digits after the decimal point. Generally we aren't interested in seeing so many digits and we particularly aren't interested in this case because our computations aren't that accurate.

Before we try to complete the code for the table, let's think about the problem of displaying only some of the digits of a number. The idea is to truncate the digits so that we don't have to see all of them. We've seen a way to truncate digits. We can cast a double to an int and that truncates all of the digits after the decimal point. We could do that, but we probably want at least some of those digits. For example, how could we truncate all but two of the digits? The trick is to bring the two digits we want to the other side of the decimal point. We can do that by multiplying by 100 and then casting to int:

```
(int) (n * 100.0)
```

This expression gets us the digits we want, but now the decimal point is in the wrong place. For example, if n is intially 3.488834, the expression above will give us 348. We have to divide this result by 100 to turn it back into the number 3.48:

```
(int) (n * 100.0) / 100.0
```

While we're at it, we can make one final improvement. Notice how our original number was 3.488834. If we do simple truncation, we get 3.48. But really this number is closer to 3.49. We can round to the nearest digit by adding 0.5 to the number before we cast:

```
(int) (n * 100.0 + 0.5) / 100.0
```

This is an operation that we are likely to want to perform on more than one number, so it deserves to be included in a method:

```
public static double round2(double n) {
    return (int) (n * 100.0 + 0.5) / 100.0;
}
```

Getting back to our pseudocode for the table, we can incorporate calls on the round2 method to get a bit closer to actual Java code:

```
set all of x, y and t to 0
for (given number of steps) {
    t += timeIncrement;
    x += xIncrement;
    y = yVelocity * t + 0.5 * EARTH_ACCELERATION * t * t;
    report step #, round2(x), round2(y), round2(t)
}
```

It would be nice if the values in the table lined up somewhat. To get numbers that line up nicely, we would have to use formatted output. We will see how to do that in Chapter 4. For now, we can at least get the numbers to line up in columns by separating them with tab characters. Remember that the escape sequence "\t" represents a single tab.

If we're going to have a table with columns, it also makes sense to have a header for the table. And we probably want to include a line in the table showing the initial condition where x, y and time are all equal to 0. So we can expand the pseudocode above into the following Java code:

```
double x = 0.0;
double y = 0.0;
double t = 0.0;
System.out.println("step\tx\ty\ttime");
System.out.println("0\t0.0\t0.0\t0.0");
for (int i = 1; i <= steps; i++) {
    t += timeIncrement;
    x += xIncrement;
    y = yVelocity * t + 0.5 * EARTH_ACCELERATION * t * t;
    System.out.println(i + "\t" + round2(x) + "\t" + round2(y) + "\t"
               + round2(t));
}
```

An Unstructured Solution

We can put all of these pieces together to form a complete program. Let's first look at an unstructured version that includes most of the code in main. This version also includes some new println statements at the beginning that give a brief introduction to the user.

```
 1  // This program computes the trajectory of a projectile.
 2
 3  import java.util.*;   // for Scanner
 4
 5  public class Projectile {
 6      public static final double EARTH_ACCELERATION = -9.81;  // meters/second^2
 7
 8      public static void main(String[] args) {
 9          Scanner console = new Scanner(System.in);
10
11          System.out.println("This program computes the trajectory of a");
12          System.out.println("projectile given its initial velocity and");
13          System.out.println("its angle relative to the horizontal.  Use");
14          System.out.println("an even number of steps if you want to include");
15          System.out.println("the high point reached by the projectile.");
16          System.out.println();
17
18          System.out.print("velocity (meters/second)? ");
19          double velocity = console.nextDouble();
20          System.out.print("angle relative to horizontal (degrees)? ");
21          double angle = Math.toRadians(console.nextDouble());
22          System.out.print("number of steps to display? ");
23          int steps = console.nextInt();
24          System.out.println();
25
26          double xVelocity = velocity * Math.cos(angle);
27          double yVelocity = velocity * Math.sin(angle);
28          double totalTime = - 2.0 * yVelocity / EARTH_ACCELERATION;
29          double timeIncrement = totalTime / steps;
30          double xIncrement = xVelocity * timeIncrement;
31
32          double x = 0.0;
33          double y = 0.0;
34          double t = 0.0;
35          System.out.println("step\tx\ty\ttime");
36          System.out.println("0\t0.0\t0.0\t0.0");
37          for (int i = 1; i <= steps; i++) {
38              t += timeIncrement;
39              x += xIncrement;
```

```
40              y = yVelocity * t + 0.5 * EARTH_ACCELERATION * t * t;
41              System.out.println(i + "\t" + round2(x) + "\t" + round2(y) + "\t"
42                                 + round2(t));
43          }
44      }
45
46      public static double round2(double n) {
47          return (int) (n * 100.0 + 0.5) / 100.0;
48      }
49  }
```

The following is a sample execution of the program.

```
This program computes the trajectory of a
projectile given its initial velocity and
its angle relative to the horizontal.  Use
an even number of steps if you want to include
the high point reached by the projectile.

velocity (meters/second)? 30
angle relative to horizontal (degrees)? 50
number of steps to display? 10

step    x       y       time
0       0.0     0.0     0.0
1       9.03    9.69    0.47
2       18.07   17.23   0.94
3       27.1    22.61   1.41
4       36.14   25.84   1.87
5       45.17   26.92   2.34
6       54.21   25.84   2.81
7       63.24   22.61   3.28
8       72.28   17.23   3.75
9       81.31   9.69    4.22
10      90.35   0.0     4.69
```

From the log, you can see that the projectile reaches a maximum height of 26.92 meters after 2.34 seconds (the fifth step) and that it lands 90.35 meters from where it began after 4.69 seconds (the tenth step).

This version of the program works, but we don't generally want to include so much code in method main. The next section explores how to break this program up into smaller pieces.

A Structured Solution

Looking at the code in the main method of the Projectile program, there are three majors blocks of code. We have a series of println statements that introduce the program to the user. Then we have a series of statements that prompt the user for a the three values used to produce the table. Then we have the code that produces the table itself.

So the overall structure looks like this:

```
give introduction.
prompt for velocity, angle and steps.
produce table.
```

The first step and the third step are easily turned into methods, but not the middle step. The middle step prompts the user for values that we need to produce the table. If we turn that middle step into a method, then it would have to somehow return three values back to main. A method can return only a single value, so we unfortunately can't turn this into a method. We could turn the code into three different methods, one for each of the three values, but each of those methods would be just two lines long, so it's not clear that it improves the overall structure much.

The main improvement we can make, then, is to split off the introduction and the table into separate methods. Another improvement we can make is to turn the Physics displacement formula into its own method. It is a good idea to turn equations into methods. Introducing those methods, we get the following structured version of the program:

```
1  // This program computes the trajectory of a projectile.
2
3  import java.util.*;   // for Scanner
4
5  public class Projectile2 {
6      public static final double EARTH_ACCELERATION = -9.81;  // meters/second^2
7
8      public static void main(String[] args) {
9          Scanner console = new Scanner(System.in);
10         giveIntro();
11
12         System.out.print("velocity (meters/second)? ");
13         double velocity = console.nextDouble();
14         System.out.print("angle relative to horizontal (degrees)? ");
15         double angle = Math.toRadians(console.nextDouble());
16         System.out.print("number of steps to display? ");
17         int steps = console.nextInt();
18         System.out.println();
19
20         printTable(velocity, angle, steps);
21     }
22
23     // prints a table showing the trajectory of an object given its initial
24     // velocity and angle and including the given number of steps in the
25     // table
26     public static void printTable(double velocity, double angle, int steps) {
27         double xVelocity = velocity * Math.cos(angle);
28         double yVelocity = velocity * Math.sin(angle);
29         double totalTime = - 2.0 * yVelocity / EARTH_ACCELERATION;
30         double timeIncrement = totalTime / steps;
31         double xIncrement = xVelocity * timeIncrement;
32
33         double x = 0.0;
34         double y = 0.0;
35         double t = 0.0;
36         System.out.println("step\tx\ty\ttime");
37         System.out.println("0\t0.0\t0.0\t0.0");
38         for (int i = 1; i <= steps; i++) {
39             t += timeIncrement;
40             x += xIncrement;
41             y = displacement(yVelocity, t, EARTH_ACCELERATION);
42             System.out.println(i + "\t" + round2(x) + "\t" + round2(y) + "\t"
43                                + round2(t));
44         }
45     }
```

```
46
47      // gives a brief introduction to the user
48      public static void giveIntro() {
49          System.out.println("This program computes the trajectory of a");
50          System.out.println("projectile given its initial velocity and");
51          System.out.println("its angle relative to the horizontal.  Use");
52          System.out.println("an even number of steps if you want to include");
53          System.out.println("the high point reached by the projectile.");
54          System.out.println();
55      }
56
57      // returns the horizontal displacement for a body given its initial
58      // velocity v, elapsed time t and acceleration a
59      public static double displacement(double v, double t, double a) {
60          return v * t + 0.5 * a * t * t;
61      }
62
63      // rounds n to 2 digits after the decimal point
64      public static double round2(double n) {
65          return (int) (n * 100.0 + 0.5) / 100.0;
66      }
67  }
```

This version executes the same way as the earlier version.

Chapter Summary

- Methods may be written to accept parameters, which are values given from the calling code into the method. Parameters allow data values to flow into the method, which can change the way the method executes. A method declared with a set of parameters can perform an entire family of similar tasks instead of exactly one task.
- When primitive values such as type `int` or `double` are passed as parameters, their values are copied into the method. Primitive parameters send values into a method, but not out of it; the method can use the data values but cannot affect the value of any variables outside of it.
- Two methods can have the same name if they declare different parameters. This is called overloading.
- Methods can be written to return a value to the calling code. This allows methods to perform a complex computation and then provide its result back to the calling code. The type of the return value must be declared in the method's header and is called the method's return type.
- Java has a class named `Math` that contains several useful static methods that you can use in your programs, such as powers, square roots and logarithms.
- An object is an entity that combines data and operations. Some objects in Java include Strings, which are sequences of text characters, and Points, which hold Cartesian (x, y) coordinates.

- Objects contain methods that implement their behavior. To call a method on an object, write its name followed by a dot, followed by the method name; for example,
`objectName.methodName();`

- A String object holds a sequence of characters. The characters have indexes starting with 0 for the first character.

- A Point object holds two `int` values x and y representing a 2-dimensional point. A point can be constructed, translated (moved to a new location), and converted into a String to be printed on the console.

- An exception is an error that occurs when a program has made an illegal action and is unable to continue executing normally.

- Java objects use reference semantics, where variables do not store actual objects but instead are simply names that refer to objects. This means that two variables can refer to the same object. If an object is modified through one of its references, the modification will also be seen in the other.

- Objects that are passed as parameters to methods behave differently from primitive parameters because the method refers to the same object that was passed in, not to a copy of the object. If the method modifies the object's data, the modification will also be seen when the method returns.

- Some programs are interactive and respond to input from the user. These programs should print a message to the user, also called a prompt, asking for the input.

- Java has a class called Scanner that reads input from the keyboard. A Scanner can read various pieces of input (also called tokens) from an input source. A Scanner can read either one token at a time or an entire line at a time.

Self-Check Exercises

Section 3.1: Parameters

1. What output is produced by the following program?

```
1  public class Odds {
2      public static void main(String[] args) {
3          printOdds(3);
4          printOdds(17 / 2);
5
6          int x = 25;
7          printOdds(37 - x + 1);
8      }
9
10     public static void printOdds(int n) {
11         for (int i = 1; i <= n; i++) {
12             int odd = 2 * i - 1;
13             System.out.print(odd + " ");
14         }
15         System.out.println();
16     }
17 }
```

2. What is the output of the following program?

```
1  public class Weird {
2      public static void main(String[] args) {
3          int number = 8;
4          halfTheFun(11);
5          halfTheFun(2 - 3 + 2 * 8);
6          halfTheFun(number);
7          System.out.println("number = " + number);
8      }
9
10     public static void halfTheFun(int number) {
11         number = number / 2;
12         for (int count = 1; count <= number; count++) {
13             System.out.print(count + " ");
14         }
15         System.out.println();
16     }
17 }
```

3. What is the output of the following program?

```
1  public class Params1 {
2      public static void main(String[] args) {
3          String one = "two";
4          String two = "three";
5          String three = "1";
6          int number = 20;
7
8          sentence(one, two, 3);
9          sentence(two, three, 14);
10         sentence(three, three, number + 1);
11         sentence(three, two, 1);
12         sentence("eight", three, number / 2);
13     }
14
15     public static void sentence(String three, String one, int number) {
16         System.out.println(one + " times " + three + " = " + (number * 2));
17     }
18 }
```

4. What output is produced by the following program?

```
1  public class Mystery1 {
2      public static void main(String[] args) {
3          String whom = "her";
4          String who = "him";
5          String it = "who";
6          String he = "it";
7          String she = "whom";
8
9          sentence(he, she, it);
10         sentence(she, he, who);
11         sentence(who, she, who);
12         sentence(it, "stu", "boo");
13         sentence(it, whom, who);
14     }
15
16     public static void sentence(String she, String who, String whom) {
17         System.out.println(who + " and " + whom + " like " + she);
18     }
19 }
```

5. Write a method named `printStrings` that accepts a String and a number of repetitions as parameters, and prints that String the given number of times. For example, the call:

   ```
   printStrings("abc", 5)
   ```

   will print the following output:

   ```
   abcabcabcabcabc
   ```

Section 3.2: Methods that Return Values

6. The System.out.println command works on many different types of values, such as int or double. What is the term for such a method?

7. What is wrong with the following program?

```
1  public class Temperature {
2      public static void main(String[] args) {
3          double tempf = 98.6;
4          double tempc = 0.0;
5          ftoc(tempf, tempc);
6          System.out.println("Body temp in C is: " + tempc);
7      }
8
9      // Converts Fahrenheit temperatures to Celsius.
10     public static void ftoc(double tempf, double tempc) {
11         tempc = (tempf - 32) * 5 / 9;
12     }
13 }
```

8. Evaluate the following expressions:

   - `Math.abs(-1.6)`

- `Math.abs(2 + -4)`
- `Math.pow(6, 2)`
- `Math.pow(5 / 2, 6)`
- `Math.ceil(9.1)`
- `Math.ceil(115.8)`
- `Math.max(7, 4)`
- `Math.min(8, 3 + 2)`
- `Math.min(-2, -5)`
- `Math.sqrt(64)`
- `Math.sqrt(76 + 45)`
- `100 + Math.log10(100)`
- `13 + Math.abs(-7) - Math.pow(2, 3) + 5`
- `Math.sqrt(16) * Math.max(Math.abs(-5), Math.abs(-3))`
- `7 - 2 + Math.log10(1000) + Math.log(Math.pow(Math.E, 5))`
- `Math.max(18 - 5 + Math.random(), Math.ceil(4.6 * 3))`

9. What output is produced by the following program?

```
1  public class Mystery6 {
2      public static void main(String[] args) {
3          int x = 1, y = 2, z = 3;
4          z = mystery(x, z, y);
5          System.out.println(x + " " + y + " " + z);
6          x = mystery(z, z, x);
7          System.out.println(x + " " + y + " " + z);
8          y = mystery(y, y, z);
9          System.out.println(x + " " + y + " " + z);
10     }
11
12     public static int mystery(int z, int x, int y) {
13         z--;
14         x = 2 * y + z;
15         y = x - 1;
16         System.out.println(y + " " + z);
17         return x;
18     }
19 }
```

10. Write a method named `min` that takes three ints as parameters and returns the value out of the three that is the smallest. A call of `min(3, -2, 7)` would return -2, and a call of `min(19, 27, 6)` would return 6. Use `Math.min` to write your solution.

11. Write a method named `countQuarters` that takes an int representing a number of cents as a parameter and returns the number of quarter coins represented by that many cents. Don't count any whole dollars, because those would be dispensed as dollar bills. For example, `countQuarters(64)` would return 2, because two quarters make 50 cents, with 14 extra left over. A call of `countQuarters(1278)` would return 3, because after the 12 dollars are taken out, 3 quarters remain in the 78 cents left.

Section 3.3: Using Objects

12. Assuming that the following variables have been declared,

```
String str1 = "Q.E.D.";
String str2 = "Arcturan Megadonkey";
String str3 = "Sirius Cybernetics Corporation";
```

Evaluate the following expressions:

- `str1.length()`
- `str2.length()`
- `str1.toLowerCase()`
- `str2.toUpperCase()`
- `str1.substring(8, 11)`
- `str2.substring(10, 14)`
- `str1.indexOf("D")`
- `str1.indexOf(".")`
- `str2.indexOf("donkey")`
- `str3.indexOf("X")`
- `str2 + str3.charAt(17)`
- `str3.substring(9, str3.indexOf("e"))`
- `str3.substring(7, 12)`
- `str2.toLowerCase().substring(9, 13) + str3.substring(18, str3.length() - 7)`

13. Based on the following String,

```
String quote = "Four score and seven years ago";
```

What expression produces the new String `"SCORE"` ? What expression produces `"four years"` ?

14. What are the x/y coordinates of the Points referred to as p1, p2, and p3 after the following code?

```
Point p1 = new Point(17, 9);
Point p2 = new Point(4, -1);
Point p3 = p2;

p1.translate(3, 1);
p2.x = 50;
p3.translate(-4, 5);
```

15. What is the output of the following program?

```
1  import java.awt.*;
2
3  public class PointMystery {
4      public static void main(String[] args) {
5          Point p1 = new Point(5, 2);
6          Point p2 = new Point(-3, 6);
7
8          silly(p1);
9          silly(p2);
10         Point p3 = p1;
11         silly(p3);
12
13         System.out.println("(" + p1.x + ", " + p1.y + ")");
14         System.out.println("(" + p2.x + ", " + p2.y + ")");
15         System.out.println("(" + p3.x + ", " + p3.y + ")");
16     }
17
18     public static void silly(Point p) {
19         int temp = p.x;
20         p.x = p.y;
21         p.y = temp;
22     }
23 }
```

Section 3.4: Interactive Programs

16. In a program reading user input using a Scanner, if the user types the following line of input:

```
Hello there.  1+2 is 3 and 1.5 squared is 2.25!
```

How many tokens are in the preceding line? What types can the Scanner legally read each token as?

17. Given the following code fragment:

```
Scanner console = new Scanner(System.in);
System.out.print("How much money do you have? ");
double money = console.nextDouble();
```

Describe what will happen when the user types each of the following values. If the code will run successfully, describe the value that will be stored in the variable `money`.

- 34.50
- 6
- $25.00
- million
- 100*5
- 600,000
- none
- 645.

18. Write Java code to read an integer from the user, then print that number multiplied by 2. You may assume that the user types a valid integer.

19. Write Java code that prompts the user for a phrase and a number of times, then prints the phrase that many times. Here is an example dialogue with the user:

```
What is your phrase? His name is Robert Paulson.
How many times should I repeat the phrase? 3

His name is Robert Paulson.
His name is Robert Paulson.
His name is Robert Paulson.
```

Exercises

1. Write a method named `printNumbers` that accepts a maximum number as an argument and prints each number from 1 up to that maximum, inclusive, boxed by square brackets. For example, consider the following calls:

```
printNumbers(15);
printNumbers(5);
```

   The preceding calls should produce the following output:

```
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15]
[1] [2] [3] [4] [5]
```

   You may assume that the value passed to `printNumbers` has a value of 1 or greater.

2. Write a method named `printPowersOf2` that accepts a maximum number as an argument and prints each power of 2 from 2^0 (1) up to that maximum power, inclusive. For example, consider the following calls:

```
printPowersOf2(3);
printPowersOf2(10);
```

   The preceding calls should produce the following output:

```
1 2 4 8
1 2 4 8 16 32 64 128 256 512 1024
```

   You may assume that the value passed to `printPowersOf2` has a value of 0 or greater. (The `Math` class may help you with this problem. If you use it, you may need to cast its results from double to int so that you don't see a .0 after each number in your output. Also, can you write this program without using the `Math` class?)

3. Write a method named `printPowersOfN` that accepts a base and an exponent as arguments and prints each power of the base from base^0 (1) up to that maximum power, inclusive. For example, consider the following calls:

```
printPowersOfN(4, 3);
printPowersOfN(5, 6);
printPowersOfN(-2, 8);
```

The preceding calls should produce the following output:

```
1 4 16 64
1 5 25 125 625 3125 15625
1 -2 4 -8 16 -32 64 -128 256
```

You may assume that the exponent passed to `printPowersOfN` has a value of 0 or greater. (The `Math` class may help you with this problem. If you use it, you may need to cast its results from double to int so that you don't see a .0 after each number in your output. Also, can you write this program without using the `Math` class?)

4. Write a method named `largerAbsVal` that takes two ints as parameters and returns the larger of the two absolute values. A call of `largerAbsVal(11, 2)` would return 11, and a call of `largerAbsVal(4, -5)` would return 5.

5. Write a variation of the method from the last exercise called `largestAbsVal` that takes three ints returns the largest of their three absolute values. For example, a call of `largestAbsVal(7, -2, -11)` would return 11, and a call of `largestAbsVal(-4, 5, 2)` would return 5.

6. Write a method named `quadratic` that solves quadratic equations and prints their roots. Recall that a quadratic equation is a polynomial equation in terms of a variable x of the form $ax^2 + bx + c = 0$. The formula for solving a quadratic equation is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Here are some example equations and their roots:

$x^2 - 7x + 12$: $x = 4$, $x = 3$
$x^2 + 3x + 2$: $x = -2$, $x = -1$

Your method should accept the coefficients a, b, and c as parameters and should print the roots of the equation. You may assume that the equation has two real roots, though mathematically this is not always the case.

7. Write a method named `padString` that accepts two parameters: a String and an integer representing a length. The method should pad the parameter string with spaces until its length is the given length. For example, `padString("hello", 8)` should return `" hello"`. (This sort of method is useful when trying to print output that lines up horizontally.) If the string's length is already at least as long as the length parameter, your method should return the original string. For example, `padString("congratulations", 10)` should return `"congratulations"`.

8. Write a method named `vertical` that accepts a String as its parameter and prints each letter of the string on separate lines. For example, a call of `vertical("hey now")` should produce the following output:

```
h
e
y

n
o
w
```

9. Write a method named `reverse` that accepts a String as its parameter and returns a String that is the parameter reversed, with the characters in opposite order. For example, a call of `reverse("hello there!")` should return `"!ereht olleh"`. If the empty string is passed to `reverse`, the empty string should be returned.

10. Write a method named `swapPoints` that takes two Points as arguments and swaps their values with each other. Consider the following example code that calls `swapPoints`:

```
Point p1 = new Point(5, 2);
Point p2 = new Point(-3, 6);

swapPoints(p1, p2);

System.out.println("(" + p1.x + ", " + p1.y + ")");
System.out.println("(" + p2.x + ", " + p2.y + ")");
```

    The output produced from the above code should be:

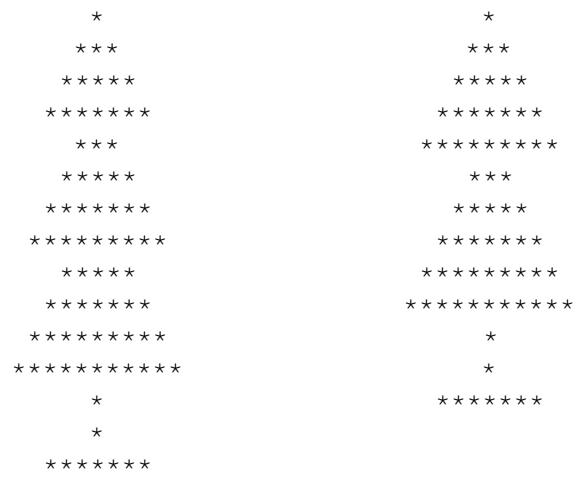
```
(-3, 6)
(5, 2)
```

11. Write Java code that prompts the user to enter his/her full name, then prints the name in the opposite order. Here is an example dialogue with the user:

```
Please enter your full name: Sammy Jankis
Your name in reverse order is Jankis, Sammy
```

Programming Projects

1. Write a program that produces Christmas trees as output. It should have a method with two parameters: one for the number of segments in the tree and one for the height of each segment. For example, the following tree on the left has 3 segments of height 4 and the one on the right has 2 segments of height 5.



2. A certain bank offers 6.5% interest on loans compounded annually. Create a table that shows how much money a person will accumulate over a period of 25 years assuming an initial investment of $1000 and assuming that $100 is deposited each year after the first. Your table should indicate for each year the current balance, the interest, the new deposit, and the new balance.

3. Write a program that shows the total number of presents received on each day according to the song "The Twelve Days of Christmas," as indicated in the following table:.

Day	Presents Received	Total Presents
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
...	...	...

4. Write a program that prompts for the lengths of the sides of a triangle and reports the three angles.

5. Write a program that computes the spherical distance between two points on the surface of the Earth. This is a useful operation because it tells you how apart two cities are if you multiply it by the radius of the Earth, which is roughly 6372.795 km.

   Let φ_1, λ_1, φ_2, λ_2 be the latitude and longitude of two points, respectively. $\Delta\lambda$ the longitude difference and $\Delta\sigma$ the angular difference/distance in radians can be determined from the spherical law of cosines as:

   $$\Delta\lambda = \arccos(\sin \varphi_1 \sin \varphi_2 + \cos \varphi_1 \cos \varphi_2 \cos \Delta\sigma)$$

   For example, consider the latitude and longitude of two major cities:

   - Nashville, TN, USA: N 36°7.2', W 86°40.2

- Los Angeles, CA, USA: N 33°56.4', W 118°24.0'

You must convert these coordinates to radians before you can use them effectively in the formula. After conversion, the coordinates become:

- Nashville: φ_1 = 36.12° = 0.6304 rad, λ_1 = -86.67° = -1.5127 rad
- Los Angeles: φ_2 = 33.94° = 0.5924 rad, λ_2 = -118.40° = -2.0665 rad

Using these values in the angular distance equation:

$$r \Delta\sigma = 6372.795 \times 0.45306 = 2887.259 \text{ km}$$

Thus the distance between these cities is about 2887 km or 1794 miles. (Note: To solve this problem, you will need to use the `Math.acos` method that returns an arc-cosine angle in radians.)

Stuart Reges
Marty Stepp

Supplement 3G
Graphics (optional)

Copyright © 2006 by Stuart Reges and Marty Stepp

- 3G.1 Introduction to Graphics
  - DrawingPanel
  - Lines and Shapes
  - Colors
  - Text and Fonts
- 3G.2 Procedural Decomposition with Graphics
  - A Larger Example: DrawDiamonds
  - Summary of Graphics Methods
- 3G.3 Case Study: Pyramids
  - An Unstructured Solution
  - A Structured Solution

Introduction

One of the most compelling reasons to learn about using objects is that they allow us to draw graphics in Java. Graphics are used for games, rendering complex images, computer animation, and modern user interfaces called GUIs (Graphical User Interfaces). In this optional supplement section we will examine a few of the basic classes from Java's graphical framework and use them to draw patterned two-dimensional figures of shapes and text onto the screen.

3G.1 Introduction to Graphics

Java's original graphical tools were collectively known as AWT which is short for "Abstract Windowing Toolkit." The classes associated with AWT reside in package java.awt. As we saw earlier in Chapter 3, the Point class is defined in this package. In order to create graphical programs like those seen in this section, we'll need the following import statement at the top of our programs.

```
import java.awt.*;   // for graphics
```

To keep things simple, we will be using a custom class called DrawingPanel that was written by the authors of this textbook to simplify some of the more esoteric details of Java graphics. It is less than a page in length, so we aren't hiding much. DrawingPanel does not need to be imported, but the file DrawingPanel.java must be placed into the same folder as your program.

The actual drawing will be done with an object of type Graphics. The Graphics class is part of the java.awt library. The drawing panel is like a piece of paper and the Graphics object is like a pencil or paintbrush. The panel keeps track of the overall image, but the Graphics object does the actual drawing of individual shapes within the panel. We will later explore the option of using different colors using Color objects.

DrawingPanel

You can create a graphical window on your screen by constructing a DrawingPanel object. You must specify the width and height of the drawing area:

```
DrawingPanel <name> = new DrawingPanel(<width>, <height>);
```

Once you have constructed one, a DrawingPanel object has the following public methods:

Useful Methods of DrawingPanel Objects

Method	Description
`getGraphics()`	returns a reference to the Graphics object that can be used to draw onto the panel
`setBackground(Color c)`	sets the background color of the panel to the given color (default is white)

The typical usage of the DrawingPanel will be to construct one with a particular height and width, to set its background color if you don't want the default white background, and then to draw something using its Graphics object. The DrawingPanel appears on the screen at the time that you construct it.

All coordinates are specified as integers. Each (x, y) position corresponds to a different pixel on the computer screen. The word *pixel* is a shorthand for "picture element" and represents a single dot on the computer screen.

The coordinate system assigns the upper-left corner of a panel to (0, 0). As we move to the right of this position, the x value increases. As we go down from this position, the y value increases. For example, suppose that you construct a DrawingPanel with a width of 200 pixels and a height of 100 pixels. Then the upper-left corner has coordinates (0, 0) and the lower-right corner has coordinates (199, 99).

```
(0, 0) +------------+
       |            |
       |            |
       |            |
       +------------+ (199, 99)
```

This is likely to be confusing at first because in math classes you probably used coordinate systems where y values went down as you moved down.

Lines and Shapes

So how do you actually draw something? To draw shapes and lines, we don't talk directly to the DrawingPanel but instead to a related object of a type named Graphics. Think of the DrawingPanel as a canvas and the Graphics object as the paintbrush. The DrawingPanel class has a method named `getGraphics` that returns a reference to its Graphics object. One of the simplest drawing commands is drawLine which takes four integer arguments:

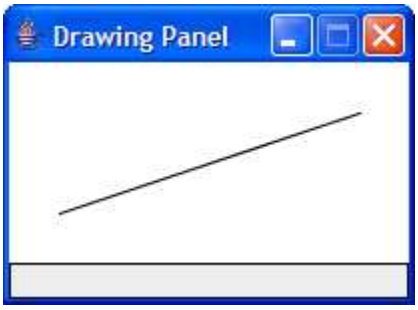
```
g.drawLine(<x1>, <y1>, <x2>, <y2>);
```

draws a line from the point (x1, y1) to the point (x2, y2)

Here is a sample program that puts these pieces together:

```
1  import java.awt.*;   // for Graphics, Color, etc
2
3  public class DrawSample1 {
4      public static void main(String[] args) {
5          // create the drawing panel
6          DrawingPanel panel = new DrawingPanel(200, 100);
7
8          // draw a line on the panel using the Graphics paintbrush
9          Graphics g = panel.getGraphics();
10         g.drawLine(25, 75, 175, 25);
11     }
12 }
```

which produces the following output:



The first statement in main constructs a DrawingPanel with a width of 200 and a height of 100. Once constructed, the window will pop up on the screen. The second statement draws a line from (25, 75) to (175, 25). The first point is in the lower-left part of the window (25 over from the left, 75 down from the top). The second point is in the upper-right corner (175 over from the left, only 25 down from the top).

Notice these particular lines of code:

```
Graphics g = panel.getGraphics();
g.drawLine(25, 75, 175, 25);
```

You might wonder why you can't just say:

```
panel.drawLine(25, 75, 175, 25);   // this is illegal
```

The point is that there are two different objects involved in this program: the drawing panel itself (the canvas) and the Graphics object associated with the panel (the paintbrush). The panel doesn't know how to draw a line. Only the Graphics object knows how to draw a line. You have to be careful to make sure that you are talking to the right object when you give a command.

This can be confusing, but it is a common occurrence in Java programs. In fact, a typical Java program has hundreds if not thousands of objects that are interacting with each other. This isn't so unlike what we do as people. If you want to schedule a meeting with a busy corporate executive, she might tell you to, "Talk to my secretary about that." Or if you're asking difficult legal questions, a person might tell you to, "Talk to my lawyer about that." In this case, the DrawingPanel doesn't know how to draw. If it could talk, it would say, "Talk to my Graphics object about that."

It would also have been legal to use the Graphics without storing it in a variable, like this:

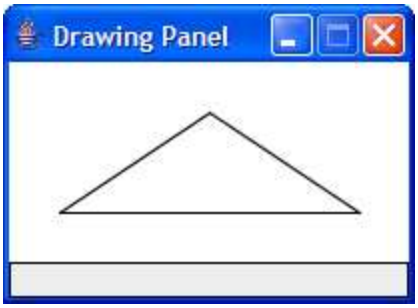
```
panel.getGraphics().drawLine(25, 75, 175, 25);   // this is also legal
```

But in many cases we'll want to send several commands to the Graphics object, so it's more convenient to give it a name and store it in a variable.

Let's look at a more complicated example:

```
 1  import java.awt.*;   // for Graphics, Color, etc
 2
 3  public class DrawSample2 {
 4      public static void main(String[] args) {
 5          DrawingPanel panel = new DrawingPanel(200, 100);
 6
 7          // draw a triangle on the panel
 8          Graphics g = panel.getGraphics();
 9          g.drawLine(25, 75, 100, 25);
10          g.drawLine(100, 25, 175, 75);
11          g.drawLine(25, 75, 175, 75);
12      }
13  }
```

This program draws three different lines to form a triangle:



The lines are drawn between three different points. In the lower-left corner we have the point (25, 75). In the middle at the top we have the point (100, 25). And in the lower-right corner we have the point (175, 75). The various calls on drawLine simply draw the lines that connect these three points.

The Graphics object also has methods for drawing particular shapes. For example, we often want to draw rectangles and we can do so with the drawRect method.

```
g.drawRect(<x>, <y>, <width>, <height>);
```

draws a rectangle with upper-left coordinates (x, y) and given height and width

Another figure we often want to draw is a circle or, more generally, an oval. We are immediately faced with the problem of how to specify where it appears and how big it is. We do so by specifying what is known as the "bounding rectangle" of the circle or oval. In other words, we specify it the same way we specify a rectangle with the understanding that Java will draw the largest oval possible that fits inside that rectangle.

```
g.drawOval(<x>, <y>, <width>, <height>);
```

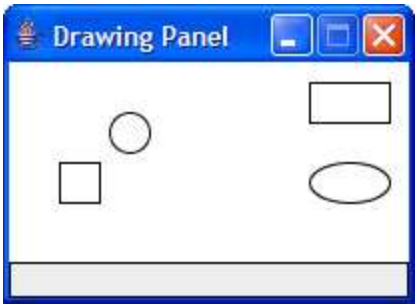
draws the largest oval that fits within the rectangle with upper-left coordinates (x, y) and given height and width

Notice that the first two values passed to drawRect and drawOval are coordinates. The next two values are a width and a height, not coordinates. For example, here is a short program that draws two rectangles and two ovals:

```
1  import java.awt.*;
2
3  public class DrawSample3 {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 100);
6
7          Graphics g = panel.getGraphics();
8          g.drawRect(25, 50, 20, 20);
9          g.drawRect(150, 10, 40, 20);
10         g.drawOval(50, 25, 20, 20);
11         g.drawOval(150, 50, 40, 20);
12     }
13 }
```

Below is the output of the program. The first rectangle has upper-left corner (25, 50). Its width and height are each 20, so this is a square. The coordinates of its lower-right corner would be (45, 70) (20 more than the (x, y) coordinates of the upper-left corner). This program also draws a rectangle with upper-left corner (150, 10) that has a width of 40 and a height of 20 (wider than it is tall). The

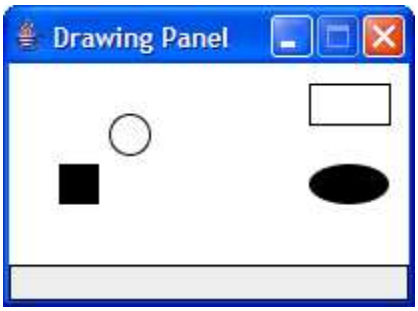
bounding rectangle of the first oval has upper-left coordinates (50, 25) and a width and height of 20. In other words, it's a circle. The bounding rectangle of the second oval has upper-left coordinates (150, 50) and a width of 40 and height of 20 (an oval that is wider than it is tall).



Sometimes we don't just want to draw the outline of a shape, we want to paint the entire area with a particular color. There are variations of the drawRect and drawOval methods known as fillRect and fillOval that do exactly this. They fill in the given rectangle or oval with the current color of paint. For example, if we change two of the calls in the previous program to be "fill" operations instead of "draw" operations:

```
1  import java.awt.*;
2
3  public class DrawSample4 {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 100);
6
7          Graphics g = panel.getGraphics();
8          g.fillRect(25, 50, 20, 20);
9          g.drawRect(150, 10, 40, 20);
10         g.drawOval(50, 25, 20, 20);
11         g.fillOval(150, 50, 40, 20);
12     }
13 }
```

we get this output instead:



Colors

All of the shapes and lines in the preceding programs were black, and all of the panels had a white background. These are the default colors. But it's possible to set the background color of the panel, and you can also change the current color being used by the Graphics object (like dipping your paintbrush in a different container of paint). We will use the standard Color class that is also part of the standard java.awt package.

There are a number of predefined colors that we can refer to directly. They are defined as class constants (just like the constants we were using in Chapter 2). The names of these constants are in all uppercase and are self-explanatory. To refer to one of these colors, you have to precede it with the class name and a dot, as in Color.GREEN or Color.BLUE. Below is a table of all of the predefined Color constants.

Color Constants

Color.BLACK	Color.GREEN	Color.RED
Color.BLUE	Color.LIGHT_GRAY	Color.WHITE
Color.CYAN	Color.MAGENTA	Color.YELLOW
Color.DARK_GRAY	Color.ORANGE	
Color.GRAY	Color.PINK	

The DrawingPanel object has a method that can be used to change the current background color that covers the entire panel:

```
<panel>.setBackground(<color>);
```

sets the panel's background to be the given color

The Graphics object has a method that can be used to change the current drawing color for shapes and lines:

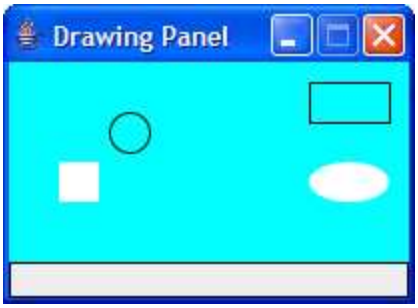
```
g.setColor(<color>);
```

sets the current drawing/filling color to be the given color

Calling setColor is like dipping your paintbrush in a different color paint. It means that from that point on, all drawing and filling will be done in that color. For example, here is another version of the program from the previous that uses a cyan (light blue) background color and fills in the oval and rectangle with white instead of black.

```
 1  import java.awt.*;
 2
 3  public class DrawSample5 {
 4      public static void main(String[] args) {
 5          DrawingPanel panel = new DrawingPanel(200, 100);
 6          panel.setBackground(Color.CYAN);
 7
 8          Graphics g = panel.getGraphics();
 9          g.drawRect(150, 10, 40, 20);
10          g.drawOval(50, 25, 20, 20);
11          g.setColor(Color.WHITE);
12          g.fillOval(150, 50, 40, 20);
13          g.fillRect(25, 50, 20, 20);
14      }
15  }
```

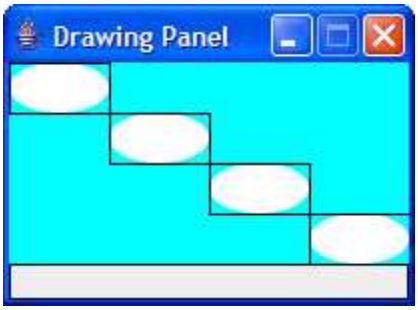
It produces the following output:



It's somewhat odd that you tell the panel to set the background color, while you tell the Graphics to set the foreground color. The reasoning is that the background color is a property of the entire window, while the foreground color only affects the particular shapes that you then draw with that color.

Notice that we have rearranged the order of the calls. The two drawing commands appear first, then the call on setColor that changes the color to white, then the two filling commands. We do things in this order so that the drawing is done in black and the filling is done in white. The order of operations is very important in these drawing programs, so you'll have to keep track of what your current color is each time you give a new command to draw or fill something.

In each of the preceding examples, we used simple constants for the drawing and filling commands. It is possible to use expressions. For example, suppose that we stick with our DrawingPanel size of 200 wide and 100 tall and we want to have a diagonal series of four rectangles that extend from the upper-left corner to the lower-right corner, each with a white oval inside. In other words, we want to produce this output:

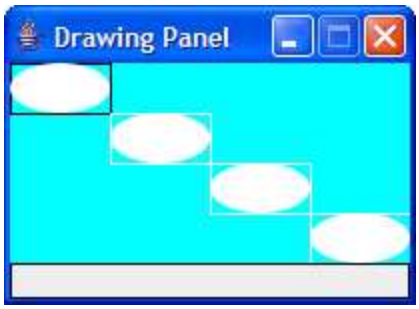


The overall width of 200 and overall height of 100 are divided evenly into four rectangles, which means that they all have to be 50 wide and 25 high. So their width and height don't change, but the position of their upper-left corners are different. The first rectangle has upper-left corner (0, 0). The second one has upper-left corner (50, 25). The third has upper-left corner (100, 50). And the fourth has upper-left corner (150, 75). We need to write code to generate these different coordinates.

This is a great place to use a for loop. Using the techniques of chapter 2, we can make a table and develop a formula for the coordinates. In this case it is easier to have the loop start with 0 rather than 1, which will often be the case with drawing programs. Below is a program that is a good first stab at generating this output.

```
1  import java.awt.*;
2
3  public class DrawSample8 {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 100);
6          panel.setBackground(Color.CYAN);
7
8          Graphics g = panel.getGraphics();
9          for (int i = 0; i < 4; i++) {
10             g.drawRect(i * 50, i * 25, 50, 25);
11             g.setColor(Color.WHITE);
12             g.fillOval(i * 50, i * 25, 50, 25);
13         }
14     }
15 }
```

It produces the following output:



We've got our coordinates and sizes right, but the colors aren't quite right. Instead of getting black rectangles with white ovals inside, we're getting one black rectangle and three white rectangles. That's because we only have one call on setColor inside the loop. Initially the color will be set to black, which is why the first rectangle comes out black. But once we make a call on setColor changing the color to white, then every subsequent drawing and filling command is done in white, including the second, third and fourth rectangles.

So we need to include a call to set the color to black for the rectangle drawing and to set the color to white for the oval. While we're at it, it's a good idea to switch the order of these things. The rectangle and oval overlap slightly and we would rather have the rectangle drawn over the oval than the other way around. Below is a program that produces the correct output.

```
1  import java.awt.*;
2
3  public class DrawSample7 {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 100);
6          panel.setBackground(Color.CYAN);
7
8          Graphics g = panel.getGraphics();
9          for (int i = 0; i < 4; i++) {
10             g.setColor(Color.WHITE);
11             g.fillOval(i * 50, i * 25, 50, 25);
12             g.setColor(Color.BLACK);
13             g.drawRect(i * 50, i * 25, 50, 25);
14         }
15     }
16 }
```

It's also possible to create custom Color objects of your own, rather than using the constant colors provided in the Color class. Computer monitors use red, green, and blue ("RGB") as their primary colors, so when you construct a Color object you pass your own parameter values for the redness, greenness, and blueness of the color.

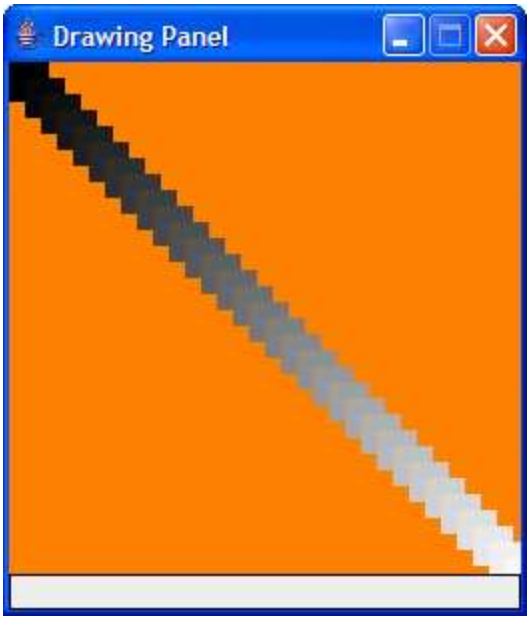
```
new Color(<red>, <green>, <blue>)
```

Each of the red/green/blue components should be integer values between 0 and 255. The higher the value, the more of that color is mixed in. All 0 values would produce black, and all 255 values would produce white. Values of (0, 255, 0) would produce fully green, while values of (128, 0, 128) make a dark purple color (because red and blue are mixed). Search for "RGB table" in your favorite search engine to find tables of many common colors.

The following program demonstrates custom colors. It uses a class constant for the number of rectangles to draw, and produces a blend of colors from black to white:

```
1  import java.awt.*;
2
3  public class DrawColorGradient {
4      public static final int RECTS = 32;
5
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(256, 256);
8          panel.setBackground(new Color(255, 128, 0));  // orange
9
10         Graphics g = panel.getGraphics();
11
12         // from black to white, top/left to bottom/right
13         for (int i = 0; i < RECTS; i++) {
14             int shift = i * 256 / RECTS;
15             g.setColor(new Color(shift, shift, shift));
16             g.fillRect(shift, shift, 20, 20);
17         }
18     }
19 }
```

It produces the following output:



Text and Fonts

There is another drawing command worth mentioning. It can be used to include text in your drawing.

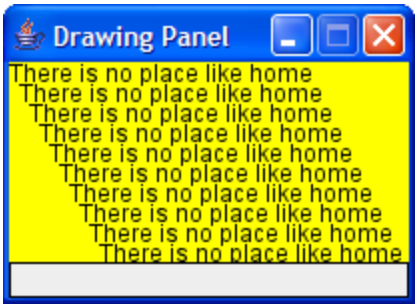
```
g.drawString(<message>, <x>, <y>);
```

draws the given String with lower-left corner (x, y)

This is a slightly different convention than what we used for drawRect. With drawRect we specified the coordinates of the upper-left corner. Here we specify the coordinates of the lower-left corner. By default, text is drawn approximately 10 pixels high. Below is a sample program that uses a loop to draw a particular String ten different times, indenting each time 5 more to the right and moving down 10 from the top.

```
1  import java.awt.*;
2
3  public class DrawStringMessage1 {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 100);
6          panel.setBackground(Color.YELLOW);
7
8          Graphics g = panel.getGraphics();
9          for (int i = 0; i < 10; i++) {
10             g.drawString("There is no place like home", i * 5, 10 + i * 10);
11         }
12     }
13 }
```

It produces the following output.



If you'd like to change the style or size in which the onscreen text is drawn, you can use the setFont command on the Graphics object.

```
g.setFont(<font>)
```

changes the text size and style in which strings are drawn

The parameter to setFont is a Font object. A Font object is constructed by passing three parameters: The font's name as a String, its style (such as bold or italic), and its size as an integer. The font styles such as bold are implemented as constants in the Font class.

```
new Font(<name>, <style>, <size>)
```

Useful Constants of the Font Class

`Font.BOLD`	**Bold text**
`Font.ITALIC`	*Italic text*
`Font.BOLD + Font.ITALIC`	***Bold/Italic text***
`Font.PLAIN`	Plain text

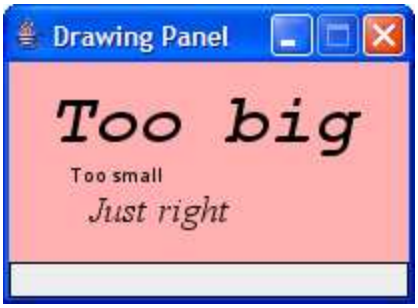
Common Font Names

`"Monospaced"`	A typewriter font, such as `Courier New`.
`"SansSerif"`	A font without curves ("serifs") at letter edges, such as Arial.
`"Serif"`	A font with curved edges, such as Times New Roman.

As with colors, setting the font only affects strings drawn after the font is set. The following program sets several fonts and uses them to draw strings.

```
1  import java.awt.*;
2
3  public class DrawFonts {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 100);
6          panel.setBackground(Color.PINK);
7
8          Graphics g = panel.getGraphics();
9          g.setFont(new Font("Monospaced", Font.BOLD + Font.ITALIC, 36));
10         g.drawString("Too big", 20, 40);
11
12         g.setFont(new Font("SansSerif", Font.PLAIN, 10));
13         g.drawString("Too small", 30, 60);
14
15         g.setFont(new Font("Serif", Font.ITALIC, 18));
16         g.drawString("Just right", 40, 80);
17     }
18 }
```

It produces the following output.

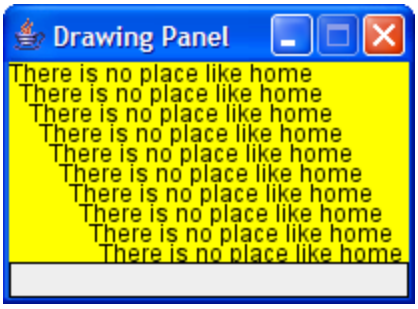


3G.2 Procedural Decomposition with Graphics

If you write complex drawing programs, you will want to break the program down into several static methods to structure the code and to remove redundancy. In doing so, you'll have to pass a reference to the Graphics object to each static method that you introduce. For a quick example, the program from the previous section could be split into a main method and a drawing method as follows.

```
1  import java.awt.*;
2
3  public class DrawStringMessage2 {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 100);
6          panel.setBackground(Color.YELLOW);
7
8          Graphics g = panel.getGraphics();
9          drawText(g);
10     }
11
12     public static void drawText(Graphics g) {
13         for (int i = 0; i < 10; i++) {
14             g.drawString("There is no place like home", i * 5, 10 + i * 10);
15         }
16     }
17 }
```

It produces the following output.



The program wouldn't compile without passing Graphics g to the drawText method because g is needed to call any drawing methods like drawString or fillRect.

A Larger Example: DrawDiamonds

Now let's consider a slightly more complicated task: drawing the largest diamond figure that will fit into a box of a particular size. The largest diamond that can fit into a box of size 50x50 looks a bit like the following diagram:

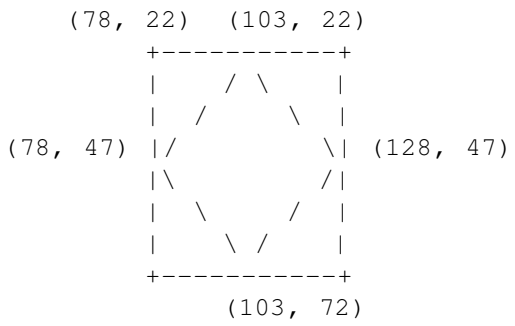
```
    (0, 0)     (25, 0)
          +------------+
          |     / \    |
          |   /     \  |
(25, 0)   |/          \| (50, 25)
          |\          /|
          |  \      /  |
          |    \ /     |
          +------------+
              (25, 50)
```

The code to draw such a diamond would be the following:

```
g.drawRect(0, 0, 50, 50);
g.drawLine(0, 25, 25, 0);
g.drawLine(25, 0, 50, 25);
g.drawLine(50, 25, 25, 50);
g.drawLine(25, 50, 0, 25);
```

Now imagine that we wish to draw 3 50x50 diamonds at different locations. We can turn our diamond drawing code into a drawDiamond method we'll call three times. Since each diamond will be in a different position, we can pass the x and y coordinates as parameters to our drawDiamond method. The parameters to the graphical calls like drawLine and drawRect will have to be changed.

A diamond with top-left corner at location (78, 22) would look like this:



The code to draw this diamond would be the following:

```
g.drawRect(78, 22, 50, 50);
g.drawLine(78, 47, 103, 22);
g.drawLine(103, 22, 128, 47);
g.drawLine(128, 47, 103, 72);
g.drawLine(103, 72, 78, 47);
```

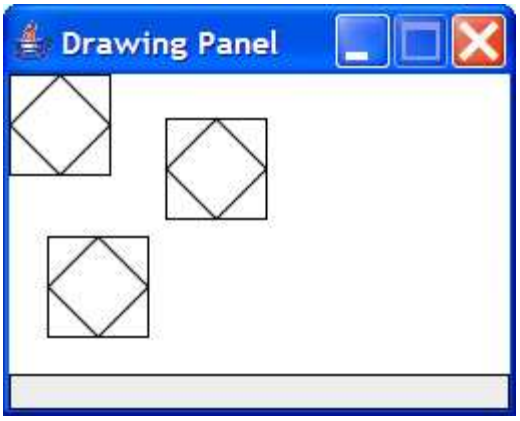
Looking at the code for the two diamonds, the parameters passed to the drawRect and drawLine commands are very similar to those of the first diamond, except that they're shifted by 78 in the x direction and 22 in the y direction (except for the third and fourth parameters to drawRect, since these are the rectangle's width and height). This (78, 22) shift is called an *offset*.

We can generalize the coordinates to pass to Graphics g's drawing commands so that they'll work with any diamond if we pass that diamond's top-left x and y offset. For example, we'll generalize the line from (0, 25) to (25, 0) in the first diamond and from (78, 47) to (103, 22) in the second diamond by saying that it is a line from $(x, y + 25)$ to $(x + 25, y)$, where (x, y) is the offset of the given diamond.

The following program uses the drawDiamond method to draw 3 diamonds without redundancy:

```
 1  // This program draws several diamond figures of size 50x50.
 2
 3  import java.awt.*;
 4
 5  public class DrawDiamonds {
 6      public static void main(String[] args) {
 7          DrawingPanel panel = new DrawingPanel(250, 150);
 8          Graphics g = panel.getGraphics();
 9
10          drawDiamond(g, 0, 0);
11          drawDiamond(g, 78, 22);
12          drawDiamond(g, 19, 81);
13      }
14
15      // draws a diamond in 50x50 box
16      public static void drawDiamond(Graphics g, int x, int y) {
17          g.drawRect(x, y, 50, 50);
18          g.drawLine(x, y + 25, x + 25, y);
19          g.drawLine(x + 25, y, x + 50, y + 25);
20          g.drawLine(x + 50, y + 25, x + 25, y + 50);
21          g.drawLine(x + 25, y + 50, x, y + 25);
22      }
23  }
```

It produces the following output.



Another challenge would be to make the size a parameter to draw diamonds of different sizes.

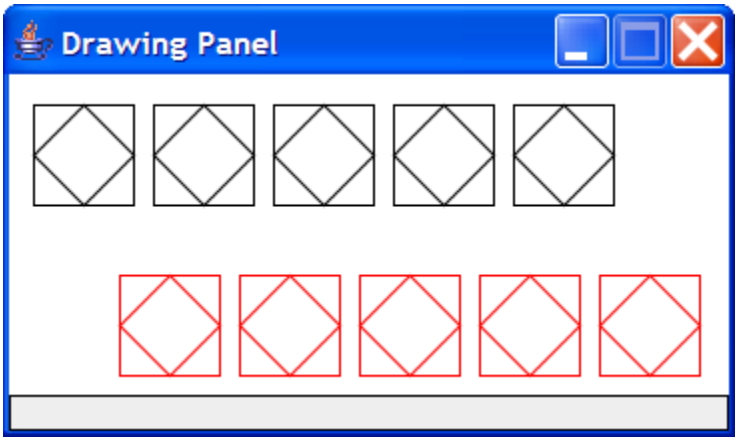
It's possible to draw patterned figures in loops and to have one of our drawing methods call another. For example, if we want to draw 5 diamonds, starting at (12, 15) and spaced 60 pixels apart, we just need a for loop that repeats 5 times and shifts the x-coordinate by 50 each time. Here's an example loop:

```
for (int i = 0; i < 5; i++) {
    drawDiamond(g, 12 + 60*i, 15);
}
```

If we wanted to turn our line of 5 diamonds into another methods, we could call it from main to draw many lines of diamonds. Here's a modified version of the DrawDiamonds program with two graphical methods:

```
1  // This program draws several diamond figures of size 50x50.
2
3  import java.awt.*;
4
5  public class DrawDiamonds2 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(360, 160);
8          Graphics g = panel.getGraphics();
9
10         drawManyDiamonds(g, 12, 15);
11         g.setColor(Color.RED);
12         drawManyDiamonds(g, 55, 100);
13     }
14
15     // Draws five diamonds in a horizontal line.
16     public static void drawManyDiamonds(Graphics g, int x, int y) {
17         for (int i = 0; i < 5; i++) {
18             drawDiamond(g, x + 60*i, y);
19         }
20     }
21
22     // draws a diamond in 50x50 box
23     public static void drawDiamond(Graphics g, int x, int y) {
24         g.drawRect(x, y, 50, 50);
25         g.drawLine(x, y + 25, x + 25, y);
26         g.drawLine(x + 25, y, x + 50, y + 25);
27         g.drawLine(x + 50, y + 25, x + 25, y + 50);
28         g.drawLine(x + 25, y + 50, x, y + 25);
29     }
30 }
```

It produces the following output.



Summary of Graphics Methods

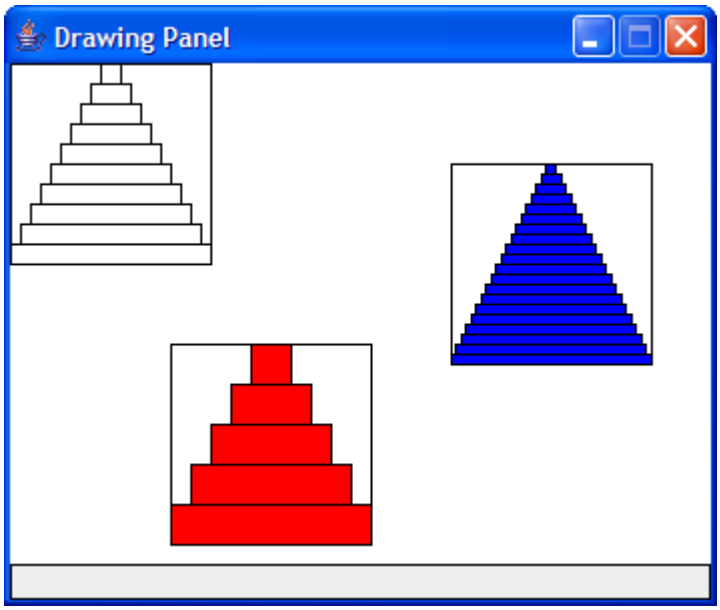
The Graphics object has many more methods than those shown in this section. If you are interested, you can read about them in the Java api documentation. Below is a list of the basic operations covered.

Some Useful Methods of Graphics Objects

Method	Description
`drawLine(x1, y1, x2, y2)`	Draws a line between the points (x1, y1) and (x2, y2).
`drawOval(x, y, width, height)`	Draws the outline of the largest oval that fits within the specified rectangle.
`drawRect(x, y, width, height)`	Draws the outline of the specified rectangle.
`drawString(message, x, y)`	Draws the given text with lower-left corner (x, y).
`fillOval(x, y, width, height)`	Fills the largest oval that fits within the specified rectangle using the current color.
`fillRect(x, y, width, height)`	Fills the specified rectangle using the current color.
`setColor(Color)`	Sets this graphics context's current color to the specified color. All subsequent graphics operations using this graphics context use this specified color.
`setFont(Font)`	Sets this graphics context's current font to the specified font. All subsequent strings drawn using this graphics context use this specified font.

3G.3 Case Study: Pyramids

Imagine that we've been asked to write a program that will draw the following figure onto a DrawingPanel:



The overall drawing panel has a size of 350 by 250. Each pyramid is 100 pixels high and 100 pixels wide. The pyramids consist of centered colored stairs that widen toward the bottom, with black outlines around each stair. Here is a table of the attributes of each pyramid:

fill color	top-left corner	# of stairs	height of each stair
white	(0, 0)	10 stairs	10 pixels
red	(80, 140)	5 stairs	20 pixels
blue	(220, 50)	20 stairs	5 pixels

An Unstructured Partial Solution

When trying to solve a larger and more complex problem like this, it's important to tackle it piece by piece and make iterative enhancements toward a final solution. Let's begin by trying to draw the top-left white pyramid correctly.

Each stair is centered horizontally within the pyramid. The top stair is 10 pixels wide. Therefore, it is surrounded by 90 / 2 or 45 pixels of empty space on either side. That means that the 10x10 rectangle's top-left corner is at (45, 0). The second stair is 20 pixels wide, meaning that it's surrounded by 80 / 2 or 40 pixels on each side.

```
(0, 0)
           +--+
<=-- 45 -=> |10| <=- 45 ---=>
          +-+--+-+
<=- 40 => |  20  | <= 40 --=>
          +------+
```

The following program draws the white pyramid at its correct position.

```
 1  import java.awt.*;
 2
 3  // Draws the first pyramid only, with a lot of redundancy.
 4  public class Pyramids1 {
 5      public static void main(String[] args) {
 6          DrawingPanel panel = new DrawingPanel(350, 250);
 7          Graphics g = panel.getGraphics();
 8
 9          // border rectangle
10          g.drawRect(0, 0, 100, 100);
11
12          // 10 "stairs" in pyramid
13          g.drawRect(45,  0,  10, 10);
14          g.drawRect(40, 10,  20, 10);
15          g.drawRect(35, 20,  30, 10);
16          g.drawRect(30, 30,  40, 10);
17          g.drawRect(25, 40,  50, 10);
18          g.drawRect(20, 50,  60, 10);
19          g.drawRect(15, 60,  70, 10);
20          g.drawRect(10, 70,  80, 10);
21          g.drawRect( 5, 80,  90, 10);
22          g.drawRect( 0, 90, 100, 10);
23      }
24  }
```

Looking at the code, it's clear that the ten lines to draw the stairs have a lot of redundancy between them. Examining the patterns of numbers in each column, the x values decrease by 5 each time, the y value increases by 10 each time, the width increases by 10 each time, and the height stays the same. Another way of describing the stair's x value is that it is half of the overall 100 minus the stair's width.

The following for loop draws the ten stairs without the previous redundancy:

```
for (int i = 0; i < 10; i++) {
    int stairWidth = 10 * (i + 1);
    int stairHeight = 10;
    int stairX = (100 - stairWidth) / 2;
    int stairY = 10 * i;
    g.drawRect(stairX, stairY, stairWidth, stairHeight);
}
```

Generalizing the Drawing of Pyramids

Next let's add code to draw the bottom red pyramid. Its x/y position is (80, 140) and it has only 5 stairs. That means each stair is twice as tall and wide as before.

```
(80, 140)
            +----+
<=- 40 -==> | 20 | <==- 40 -=>
        +---+----+---+
<= 30=> |     40     | <=30 =>
        +------------+
```

Based on this, the top stair's upper-left corner is at (120, 140) and its size is 20x20. The second stair's upper-left corner is at (110, 160) and its size is 40x20, and so on.

For the moment, let's focus on getting the coordinates of the stairs right, and not on the red fill color. Here is a redundant bit of code to draw the red pyramid's stairs, without the coloring:

```
// bottom red pyramid
g.drawRect(80, 140, 100, 100);

// 5 "stairs" of red pyramid
g.drawRect(120, 140,  20, 20);
g.drawRect(110, 160,  40, 20);
g.drawRect(100, 180,  60, 20);
g.drawRect( 90, 200,  80, 20);
g.drawRect( 80, 220, 100, 20);
```

Noticing that we again have redundancy between the five lines to draw the stairs, let's look for a pattern between them. We'll use a loop like we did in the last pyramid, but with appropriate modifications. Each stair's height is now 20 pixels, and each stair's width is now 20 times the number for that stair. The x and y coordinates are a bit trickier. The x-coordinate formula is similar to the `(100 - stairWidth) / 2` from before, but this time it must be shifted right by 80 to account for the pyramid's top-left corner. The y-coordinate must similarly be shifted downward by 140 pixels. Here's the correct loop:

```
// 5 "stairs" of red pyramid
for (int i = 0; i < 5; i++) {
    int stairWidth = 20 * (i + 1);
    int stairHeight = 20;
    int stairX = 80 + (100 - stairWidth) / 2;
    int stairY = 140 + 20 * i;
    g.drawRect(stairX, stairY, stairWidth, stairHeight);
}
```

Can we spot the pattern between these two loops? Each loop's x and y coordinates differ only by the fact that we have to add the starting point (80, 140) to the second loop. The stairs' widths and heights differ only by the fact that one pyramid's stairs are 20 pixels tall and the other's are 10. In fact, the reason they're that tall is because the overall size of 100 divided by the number of stairs (10 or 5) yields the height of each stair (10 or 20, respectively).

Using the preceding information, let's turn the code for drawing a pyramid into a method that we can call 3 times to avoid redundancy. As parameters we'll pass the (x, y) coordinates of the top-left corner of the pyramid, and the number of stairs in the pyramid. We'll also need to pass the Graphics g as a parameter so that we can draw onto the DrawingPanel. The main modification we'll make to the code is to compute the stair height first, then use this to compute the stair width, then use the width and height to help us compute the x/y coordinates of the stair. Here's the code:

```
public static void drawPyramid(Graphics g, int x, int y, int stairs) {
    // bounding rectangle
    g.drawRect(x, y, 100, 100);

    // stairs of the pyramid
    for (int i = 0; i < stairs; i++) {
        int stairHeight = 100 / stairs;
        int stairWidth = stairHeight * (i + 1);
        int stairX = x + (100 - stairWidth) / 2;
        int stairY = y + stairHeight * i;
        g.drawRect(stairX, stairY, stairWidth, stairHeight);
    }
}
```

The preceding code is now generalized to draw a pyramid at any location with any number of stairs. One final ingredient is missing: the ability to give a different color to each pyramid.

A Complete Structured Solution

The preceding code is correct except that it doesn't allow us to draw the pyramid in the proper color. Let's add an additional parameter, a Color, to our method and use it to fill the pyramid stairs as needed. We'll pass Color.WHITE as this parameter's value for the first white pyramid; it'll fill the stairs with white, even though it doesn't have to.

The way to draw a filled shape with an outline of a different color is to first fill the shape, then use the outline color to draw the same shape. For example, to get red rectangles with black outlines, first we'll use fillRect with red, then we'll use drawRect with black with the same parameters.

Here's the new version of the drawPyramid method that uses the fill color as a parameter:

```
public static void drawPyramid(Graphics g, Color c,
        int x, int y, int stairs) {
    g.drawRect(x, y, 100, 100);

    for (int i = 0; i < stairs; i++) {
        int stairHeight = 100 / stairs;
        int stairWidth = stairHeight * (i + 1);
        int stairX = x + (100 - stairWidth) / 2;
        int stairY = y + stairHeight * i;

        g.setColor(c);
        g.fillRect(stairX, stairY, stairWidth, stairHeight);
        g.setColor(Color.BLACK);
        g.drawRect(stairX, stairY, stairWidth, stairHeight);
    }
}
```

Using this method, we can now draw all 3 pyramids easily by calling drawPyramid 3 times with the appropriate parameters.

```
drawPyramid(g, Color.WHITE, 0, 0, 10);
drawPyramid(g, Color.RED, 80, 140, 5);
drawPyramid(g, Color.BLUE, 220, 50, 20);
```

One last improvement we can make to our Pyramids program is to turn the overall pyramid size of 100 into a constant, so there aren't so many 100s lying around in the code. Here is the complete program:

```
 1  // This program uses DrawingPanel to draw three colored
 2  // pyramid figures onto the screen.
 3
 4  import java.awt.*;
 5
 6  public class Pyramids {
 7      public static final int SIZE = 100;
 8
 9      public static void main(String[] args) {
10          DrawingPanel panel = new DrawingPanel(350, 250);
11          Graphics g = panel.getGraphics();
12
13          drawPyramid(g, Color.WHITE, 0, 0, 10);
14          drawPyramid(g, Color.RED, 80, 140, 5);
15          drawPyramid(g, Color.BLUE, 220, 50, 20);
16      }
17
18      // Draws one pyramid figure with the given number of
19      // stairs at the given x/y position with the given color.
20      public static void drawPyramid(Graphics g, Color c,
21              int x, int y, int stairs) {
22
23          // bounding rectangle
24          g.drawRect(x, y, SIZE, SIZE);
25
26          // stairs of the pyramid
27          for (int i = 0; i < stairs; i++) {
28              int stairHeight = SIZE / stairs;
29              int stairWidth = stairHeight * (i + 1);
30              int stairX = x + (SIZE - stairWidth) / 2;
```

```
31              int stairY = y + stairHeight * i;
32
33              // fill rectangle with fill color
34              g.setColor(c);
35              g.fillRect(stairX, stairY, stairWidth, stairHeight);
36
37              // draw black rectangle outline
38              g.setColor(Color.BLACK);
39              g.drawRect(stairX, stairY, stairWidth, stairHeight);
40          }
41      }
42  }
```

Chapter Summary

- DrawingPanel is a custom class provided by the authors to easily show a graphical window on the screen. A DrawingPanel contains a Graphics object, which can be used to draw lines, text, and shapes on the screen using different colors.
- A Graphics object has many useful methods for drawing shapes and lines, such as drawLine, fillRect, and setColor. Shapes can be "drawn" (only drawing their outline) or "filled" (coloring the entire shape).
- The Graphics object can write text on the screen with its drawString method. Different font styles and sizes can be specified with the setFont method.
- Graphical programs that are decomposed into methods must pass appropriate parameters. The Graphics object must be passed, as well as any x/y coordinates, sizes, or other values that guide the figures to be drawn.

Self-Check Exercises

Section 3G.1: Introduction to Graphics

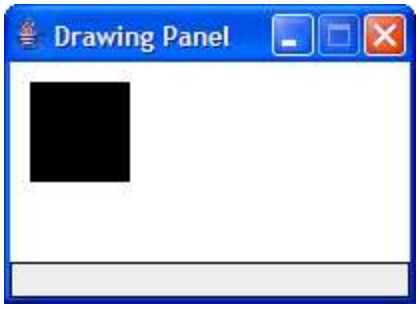
1. There are two mistakes in the following code attempting to draw a line from coordinates (50, 86) to (20, 35). What are they?

```
DrawingPanel panel = new DrawingPanel(200, 200);
panel.drawLine(50, 20, 86, 35);
```

2. The following code attempts to draw a filled black outer rectangle with a white filled inner circle inside it.

```
DrawingPanel panel = new DrawingPanel(200, 100);
Graphics g = panel.getGraphics();
g.setColor(Color.WHITE);
g.fillOval(10, 10, 50, 50);
g.setColor(Color.BLACK);
g.fillRect(10, 10, 50, 50);
```

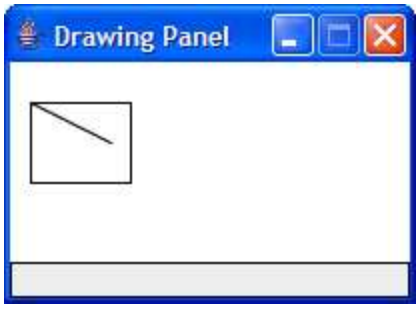
However, it has the following incorrect appearance. What must be changed for it to look correctly?



3. The following code attempts to draw a black rectangle from (10, 20) to (50, 40) with a line across its diagonal.

```
DrawingPanel panel = new DrawingPanel(200, 100);
Graphics g = panel.getGraphics();
g.drawRect(10, 20, 50, 40);
g.drawLine(10, 20, 50, 40);
```

However, it has the following incorrect appearance. What must be changed for it to look correctly?

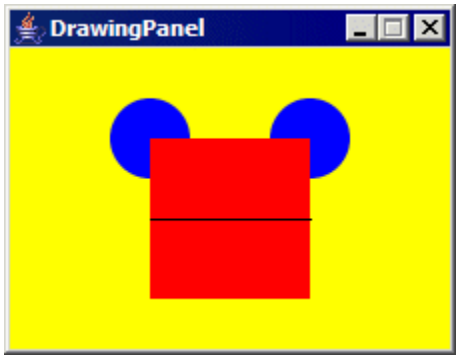


4. What sort of figure will be drawn by the following program? Can you draw an approximate picture that will match its appearance without running it first?

```
1  import java.awt.*;
2
3  public class Draw7 {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 200);
6          Graphics g = panel.getGraphics();
7          for (int i = 0; i < 20; i++) {
8              g.drawOval(i * 10, i * 10, size - (i * 10), size - (i * 10));
9          }
10     }
11 }
```

Exercises

1. Write a program that uses the DrawingPanel to draw the following figure:

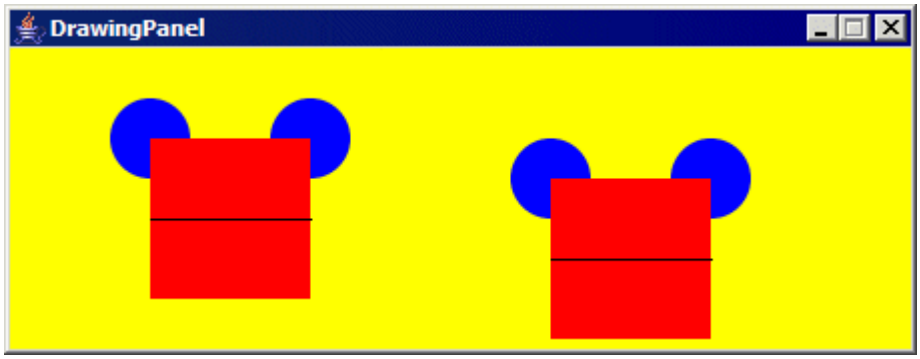


The window is 220 pixels wide and 150 pixels tall. The background is yellow. There are two blue ovals of size 40 x 40 pixels. They are 80 pixels apart, and the left oval is located at position (50, 25). There is a red square whose top two corners exactly intersect the centers of the two ovals. Lastly, there is a black horizontal line through the center of the square.

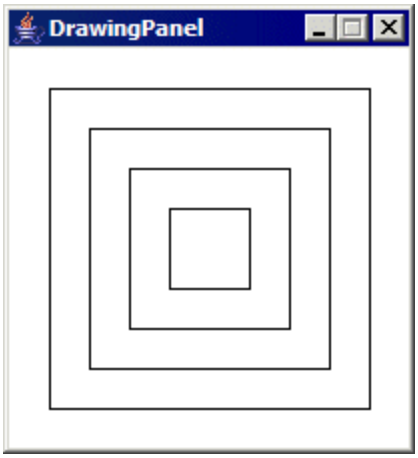
2. Modify your program from the previous exercise so that the figure is drawn by a method called `drawFigure`. The method should accept two arguments: The `Graphics g` of the DrawingPanel on which to draw, and a `Point` specifying the location of the top-left corner of the figure. Use the following heading for your method:

   ```
   public static void drawFigure(Graphics g, Point location)
   ```

   Set your DrawingPanel's size to 450 x 150 pixels, and use your `drawFigure` method to place two figures on it. One figure should be at position (50, 25) and the other should be at position (250, 45).

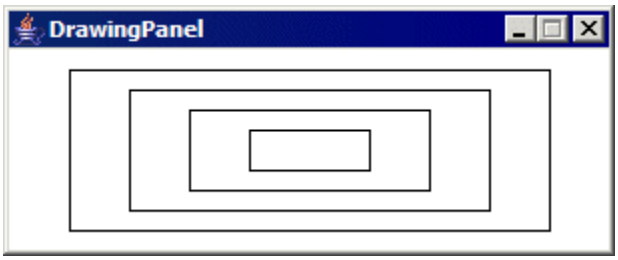


3. Write a program with a static method called `showDesign` that uses the DrawingPanel to draw the following figure:

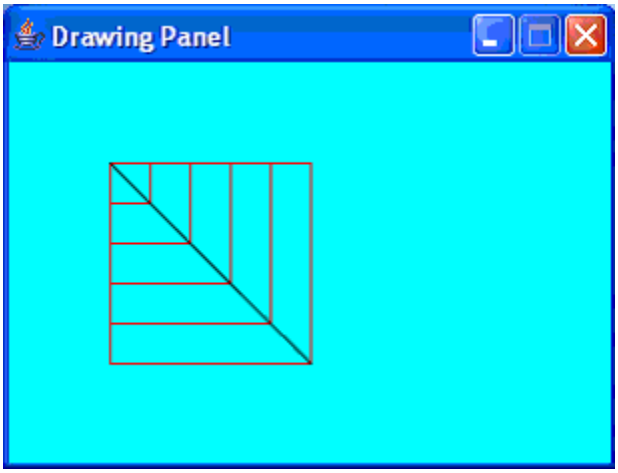


The window is 200 pixels wide and 200 pixels tall. The background is white and the foreground is black. There are 20 pixels between each of the 4 rectangles, and the rectangles are concentric (their centers are at the same point). Use a loop to draw the repeated rectangles.

4. Modify your `showDesign` method from the previous exercise so that it accepts parameters for the window width and height and displays the rectangles at the appropriate sizes. For example, if your `showDesign` method was called with values of 300 and 100, the window would have the following appearance:

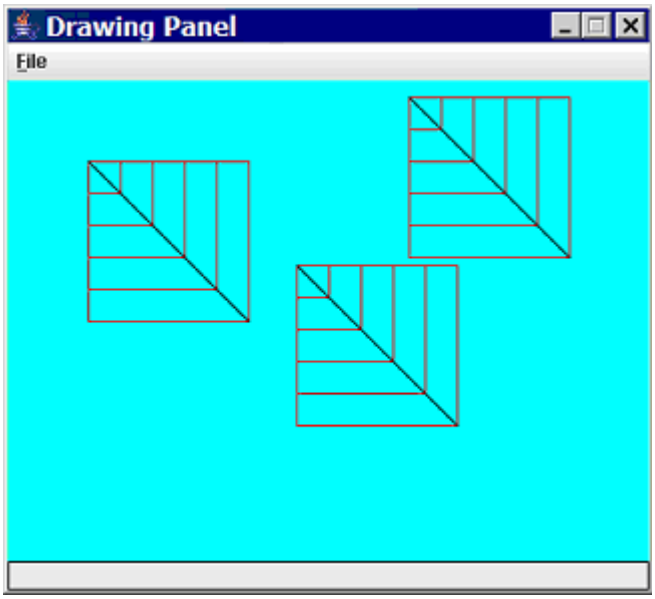


5. Write a program that uses the DrawingPanel to draw the following figure:



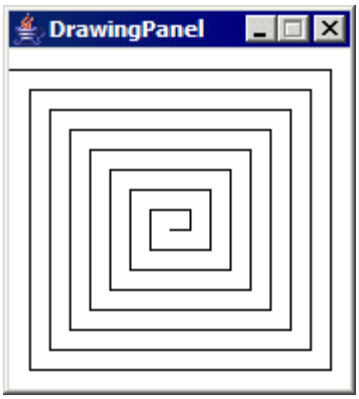
The drawing panel is 300 pixels wide by 200 pixels high. Its background is cyan. The horizontal and vertical lines are drawn in red and the diagonal line is drawn in black. The diagonal line has upper-left corner (50, 50). Successive horizontal and vertical lines are spaced 20 pixels apart.

6. Modify your code from the previous exercise to produce the following figures:



The drawing panel is now 400 x 300 pixels in size. The first figure is at the same position of (50, 50). The other figures are at positions (250, 10) and (180, 115) respectively. Use one or more parameterized static methods to reduce the redundancy of your solution.

7. Write a program that uses the DrawingPanel to draw the following spiral figure:

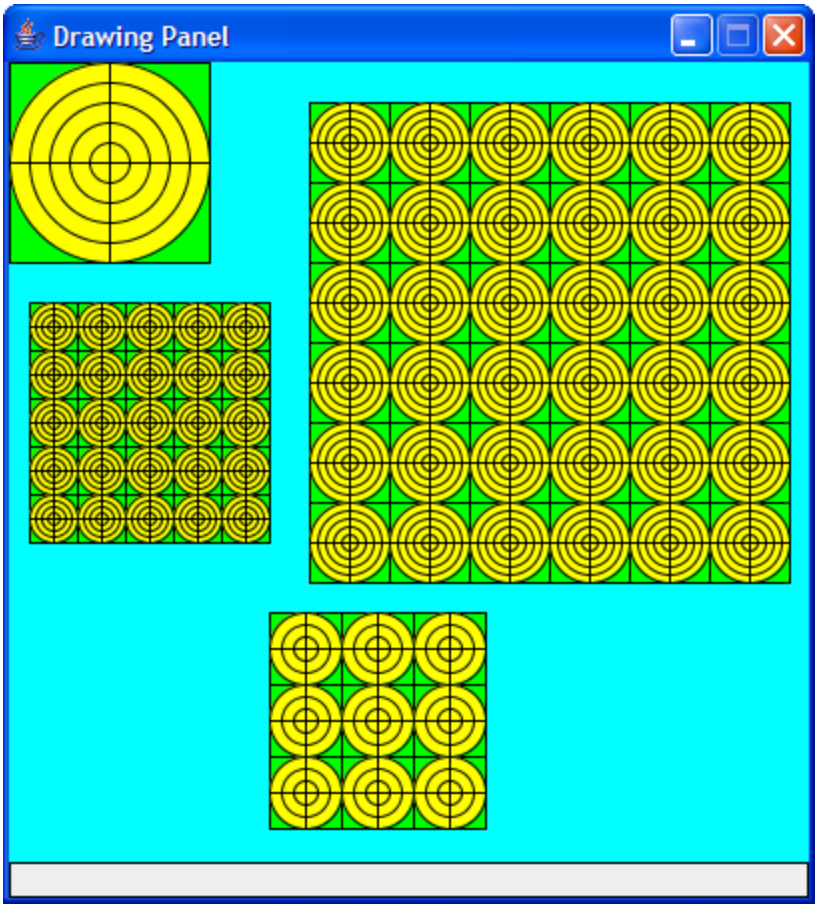


The window is 170 pixels wide and 170 pixels tall. The background is white and the foreground is black. The spiral line begins at point (0, 10) and spirals inward. There are 10 pixels between each arm of the spiral. 8 spirals are made in total. The initial spiral touches points (0, 10), (160, 10), (160, 160), (10, 160), and (10, 20).

For additional challenge, parameterize your program with parameters such as the window size and the number of spiral loops desired.

Programming Projects

1. Write a program that draws the following figure onto a DrawingPanel.

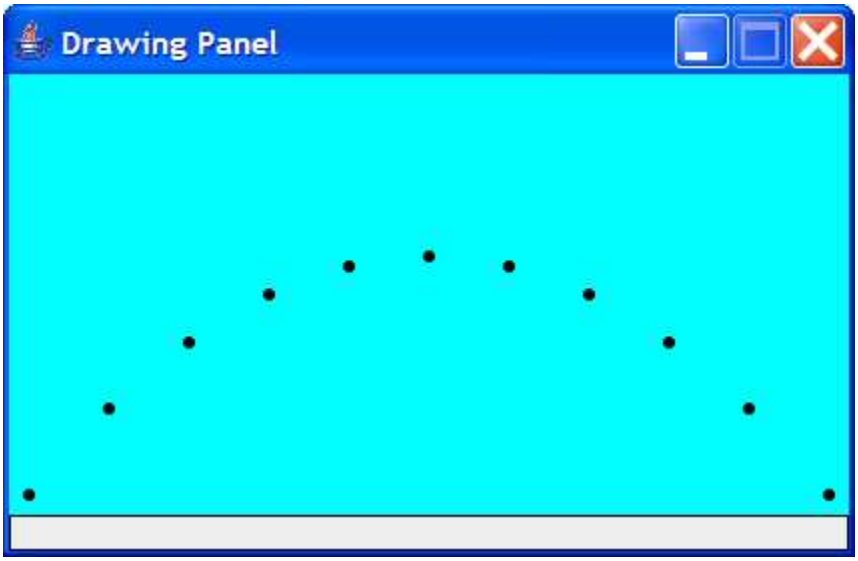


The drawing panel's size is 400x400 and its background color is cyan. It contains four figures of concentric yellow circles with black outlines, all surrounded by a green rectangle with a black outline. The four figures on your drawing panel should have the following properties:

Description	(x, y) position	size of subfigures	number of circles	number of rows/cols
top-left	(0, 0)	100 x 100	5	1 x 1
bottom-left	(10, 120)	24 x 24	4	5 x 5
top-right	(150, 20)	40 x 40	5	6 x 6
bottom-right	(130, 275)	36 x 36	3	3 x 3

Break down your program into helping methods for drawing one subfigure as well as drawing a larger grid of subfigures, such as the 5x5 grid at (10, 120).

2. Write a modified version of the Projectile case study program from Chapter 3 that draws a graph of the projectile's flight onto a DrawingPanel of size 420 x 220. For example, the following panel draws the projectile if it had an initial velocity of 30 meters per second, 50 degrees angle, and 10 steps.



Stuart Reges
Marty Stepp

Chapter 4
Conditional Execution

Copyright © 2006 by Stuart Reges and Marty Stepp

- 4.1 Loop Techniques
  - Cumulative Sum
  - Fencepost Loops (aka "loop and a half")
- 4.2 if/else Statements
  - Relational Operators
  - Cumulative Sum with if
  - Fencepost with if
  - Nested if/else
- 4.3 Subtleties of Conditional Execution
  - Object Equality
  - Roundoff Errors
  - Factoring if/else Statements
  - Min/Max Loops
- 4.4 Text Processing
  - The char Type
  - System.out.printf
- 4.5 Methods with Conditional Execution
  - Preconditions and Postconditions
  - Throwing Exceptions
  - Revisiting Return Values
- 4.6 Case Study: Body Mass Index (BMI)
  - One-person Unstructured Solution
  - Two-person Unstructured Solution
  - Two-person Structured Solution

Introduction

In the last few chapters we have seen how to solve complex programming tasks using for loops to repeat certain tasks many times. We have been able to introduce some flexibility into our programs through the use of class constants and the ability to read values from the user with a Scanner object. Now we are going to explore a much more powerful technique for writing code that can adapt to different situations.

We are going to study *conditional execution* in the form of a control structure known as the if/else. With if/else statements, we will be able to instruct the computer to execute different lines of code depending upon whether certain conditions are true. The if/else statement, like the for loop, is so powerful that you will wonder how you managed to write programs without it.

We will also be expanding our understanding of common programming situations. The chapter begins with an exploration of loop techniques we haven't yet explored and includes an exploration of text processing issues. We will also find that adding conditional execution to our repertoire will require us to revisit the issue of methods, parameters and return values so that we can better understand some of the fine points.

4.1 Loop Techniques

The more you program, the more you will find that certain patterns emerge. Before we delve into conditional execution, we are going to look at two common loop patterns that come up often in programming: cumulative sum and fencepost loops.

Cumulative Sum

We often want to find the sum of a series of numbers. You could imagine declaring a different variable for each value we want to include, but that would not be practical. If you have to add one-hundred numbers together, you don't want to have to declare one-hundred different variables. Fortunately there is a simpler way.

The trick is to keep a running tally and to process one number at a time. If you have a variable called sum, for example, you would add in the next number by saying:

```
sum = sum + next;
```

or using the shorthand assignment operator:

```
sum += next;
```

This statement says to take the old value of sum, add the value of a variable called next, and store this as the new value of sum. This operation is performed for each number to be summed. There is a slight problem when executing this statement for the first number, because there is no old value of sum the first time around. To get around this, you initialize sum to a value that will not affect the answer--zero.

Here is a pseudocode description of the cumulative sum algorithm:

```
sum = 0.
for (all numbers to sum) {
    obtain "next".
    sum += next.
}
```

To implement this algorithm, you must decide how many times to go through the loop and how to obtain a next value. Here is an interactive program that prompts the user for how many numbers to sum together and for the numbers themselves.

```
 1  // Finds the sum of a sequence of numbers.
 2
 3  import java.util.*;
 4
 5  public class ExamineNumbers1 {
 6      public static void main(String[] args) {
 7          System.out.println("This program adds a sequence of numbers.");
 8          System.out.println();
 9
10          Scanner console = new Scanner(System.in);
11
12          System.out.print("How many numbers do you want me to examine? ");
13          int totalNumber = console.nextInt();
14
15          double sum = 0.0;
16          for (int i = 1; i <= totalNumber; i++) {
17              System.out.print("    #" + i + "? ");
18              double next = console.nextDouble();
19              sum += next;
20          }
21          System.out.println();
22
23          System.out.println("sum = " + sum);
24      }
25  }
```

The program will execute something like this:

```
This program adds a sequence of numbers.

How many numbers do you want me to examine? 6
    #1? 3.2
    #2? 4.7
    #3? 5.1
    #4? 9.0
    #5? 2.4
    #6? 3.1

sum = 27.5
```

Let's trace the execution in detail. Before we enter the for loop, we initialize the variable sum to be 0:

```
    +------+
sum |  0.0 |
    +------+
```

On the first execution of the for loop, we read in a value of 3.2 from the user and add this value to the sum:

```
    +------+            +-----+
sum |  3.2 |       next | 3.2 |
    +------+            +-----+
```

The second time through the loop, we read in a value of 4.7 and add this into our sum:

```
    +------+             +-----+
sum |  7.9 |        next | 4.7 |
    +------+             +-----+
```

Notice that the sum not includes both of the numbers entered by the user because we have added the new value 4.7 to the old value of 3.2. The third time through the loop we add in the value 5.1:

```
    +------+             +-----+
sum | 13.0 |        next | 5.1 |
    +------+             +-----+
```

Notice that the variable sum now contains the sum of the first three numbers (3.2 + 4.7 + 5.1). Then we read in 9.0 and add it to the sum:

```
    +------+             +-----+
sum | 22.0 |        next | 9.0 |
    +------+             +-----+
```

Then we add in the fifth value of 2.4:

```
    +------+             +-----+
sum | 24.4 |        next | 2.4 |
    +------+             +-----+
```

And finally add in the sixth value of 3.1:

```
    +------+             +-----+
sum | 27.5 |        next | 3.1 |
    +------+             +-----+
```

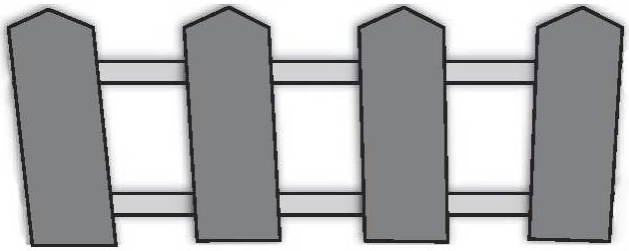
We then exit the for loop and print the value of sum. There is an interesting scope issue in this particular program. Notice that the variable sum is declared outside the loop while the variable next is declared inside the loop. We have no choice but to declare sum outside the loop because it needs to be initialized and it is used after the loop. But the variable next is used only inside the loop, so it can be declared in that inner scope. It is best to declare variables in the innermost scope possible.

The cumulative sum algorithm and variations on it will be useful in many of the programming tasks you solve. How would you do a cumulative product, for example? Here is the pseudocode:

```
product = 1.
for (all numbers to multiply) {
    obtain "next".
    product *= next.
}
```

Fencepost Loops (aka "loop and a half")

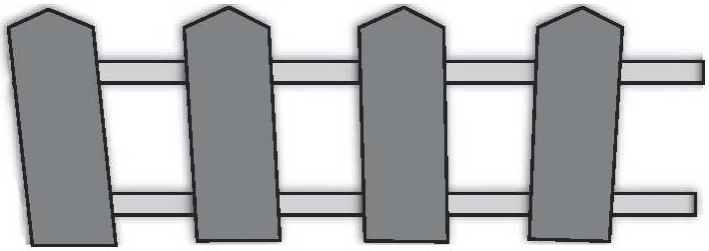
Another common programming problem involves a particular kind of loop known as a fencepost loop. Consider the following problem. You want to put up a fence that is 100 yards long and you want to have a post every 10 yards. How many posts do you need? If you do a quick division in your head, you might say that you need 10 posts, but actually you need 11 posts. That's because fences begin and end with posts. In other words, the fence looks like this:



Because you want posts on both the far left and the far right, you can't use the following simple loop because it doesn't plant the final post.

```
for (the length of the fence) {
    plant a post.
    attach some wire.
}
```

If you use the preceding loop, you'll get a fence that looks like this:



Switching the order of the two operations doesn't help, because you miss the first post. The problem with this loop is that it produces the same number of posts as sections of wire, but we know we need an extra post. That's why this problem is also sometimes referred to as the "loop and a half" problem because we want to execute one half of this loop (planting a post) one extra time.

One solution is to plant one of the posts either before or after the loop. The usual solution is to do it before.

```
plant a post.
for (the length of the fence) {
    attach some wire.
    plant a post.
}
```

Notice that the order of the two operations in the body of the loop is now reversed because the initial post is planted before you enter the loop.

As a simple example, consider the problem of writing out the integers between 1 and 10 separated by commas. In other words, we want to get this output:

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

This is a classic fencepost problem because we want to write out 10 numbers but only 9 commas. In our fencepost terminology, writing a number is the "post" part of the task and writing a comma is the "wire" part. So implementing the pseudocode above, we print the first number before the loop:

```
System.out.print(1);
for (int i = 2; i <= 10; i++) {
    System.out.print(", " + i);
}
System.out.println();
```

4.2 if/else Statements

You will often find yourself writing code that you want to execute some of the time but not all of the time. For example, if you are writing a game-playing program and a new high score has been reached, you might want to print a message and remember the new high score. You can accomplish this by putting the two lines of code inside an if statement:

```
if (currentScore > maxScore) {
    System.out.println("A new high score!");
    maxScore = currentScore;
}
```

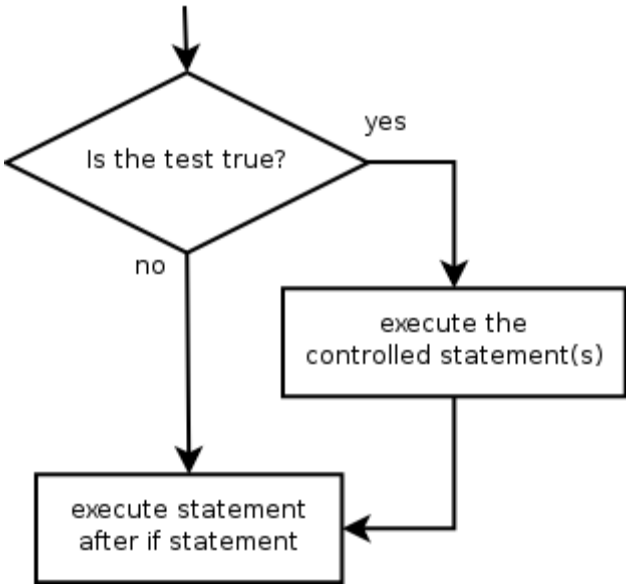
The idea is that we sometimes want to execute the two lines of code inside the if, but not always. The test in parentheses determines whether or not the statements inside the if are executed. In other words, the test describes the conditions under which we want to execute the code.

The general form of the if statement is as follows:

```
if (<test>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

The if statement, like the for loop, is a control structure. Notice that we once again see a Java keyword ("if") followed by parentheses and a set of curly braces enclosing a series of controlled statements.

The diagram below indicates the flow of control for the simple if statement. The computer performs the test and if it evaluates to true, the computer executes the controlled statements. If the test evaluates to false, the computer skips the controlled statements.



We use the simple if statement when we have code that we sometimes want to execute and sometimes want to skip. Java has a variation known as the if/else statement that allows us to choose between two alternatives. Suppose, for example, that we want to set a variable called answer to the square root of a number:

```
answer = Math.sqrt(number);
```

We have a potential problem in that we don't want to ask for the square root if the number is negative. We could use a simple if statement to avoid the problem:

```
if (number >= 0) {
    answer = Math.sqrt(number);
}
```

This will avoid asking for the square root of a negative number, but then what value will answer have if number is negative? This is a case where we would probably want to give a value to answer either way. Suppose we want answer to be -1 when number is negative. We can express this pair of alternatives with an if/else statement:

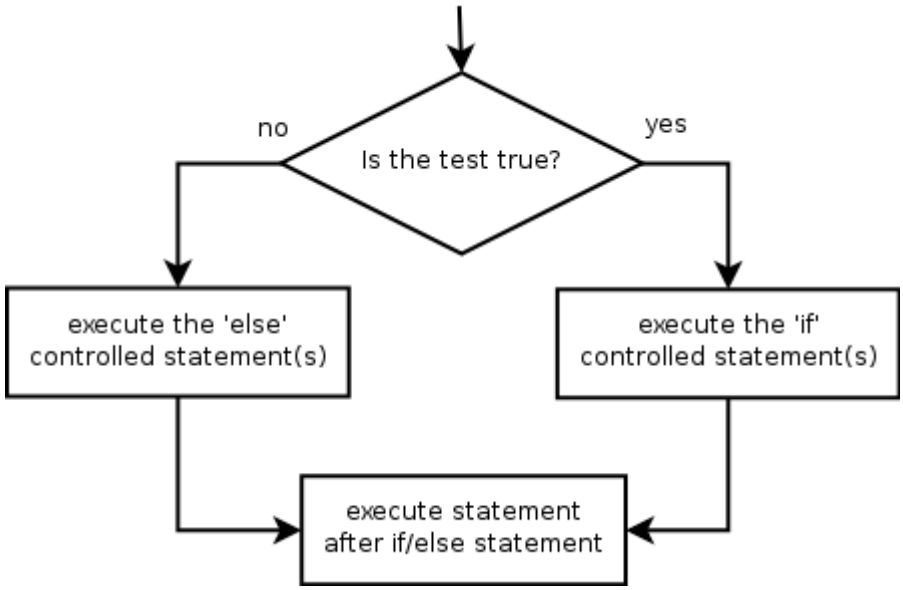
```
if (number >= 0) {
    answer = Math.sqrt(number);
} else {
    answer = -1;
}
```

The idea behind the if/else is that we choose between two alternatives and execute one or the other. So in the code above, we know that answer will be assigned a value one way or the other.

The general form of the if/else statement is as follows:

```
if (<test>) {
    <statement>;
    <statement>;
    ...
    <statement>;
} else {
    <statement>;
    ...
    <statement>;
    <statement>;
}
```

This control structure is unusual in that it has two sets of controlled statements and two different keywords (if and else). The diagram below indicates the flow of control. The computer performs the test and depending upon whether it evaluates to true or false, the computer executes one or the other groups of statements.



Thus, the if/else controls two different sets of statements, one to be executed when the test evaluates to true and the other to be executed when the test evaluates to false.

As with the for loop, if you have a single statement to execute, then you don't need to include the curly braces. But the Sun convention is to include the curly braces even if you don't need them and we follow that convention in this book.

Relational Operators

The if/else statements are controlled by a test. Simple tests compare two expressions to see if they are related in some way. Such a test is itself an expression that returns either true or false and is of the following form:

```
<expression>  <relational operator>  <expression>
```

To evaluate such a test, you first evaluate the two expressions and then see if the given relation holds between the value on the left and the value on the right. If the relation does hold, the test evaluates to true. If not, the test evaluates to false.

The relational operators are:

Relational Operators

Operator	Meaning	Example	Value
==	equals	2 + 2 == 4	true
!=	not equals	3.2 != 4.1	true
<	less than	4 < 3	false
>	greater than	4 > 3	true
<=	less than or equals	2 <= 0	false
>=	greater than or equals	2.4 >= 1.6	true

Notice that the test for equality involves two equals characters in a row ("=="). This is to distinguish it from the assignment operator ("=").

Because we use the relational operators as a new way of forming expressions, we must reconsider precedence. Below is an updated version of the precedence table that includes these new operators. You will see that technically the equality comparisons are considered at a slightly different level of precedence than the other relational operators, but both sets of operators have a lower precedence than the arithmetic operators.

Java Operator Precedence

Description	Operators
unary operators	++, --, +, -
multiplicative operators	*, /, %
additive operators	+, -
relational operators	<, >, <=, >=
equality operators	==, !=
assignment operators	=, +=, -=, *=, /=, %=

The following expression is made up of the constants 3, 2, 9, and the operations plus, times and equals.

```
3 + 2 * 2 == 9
```

Which of the operations is performed first? Because the relational operators have a lower level of precedence than the arithmetic operators, the answer is that the times is performed first, then the plus, then the "equals" test. In other words, Java will perform all of the "math" first before it tests for one of these relationships. This precedence scheme frees you from parenthesizing the left and right sides of a test using a relational operator. Using these precedence rules, the expression above is evaluated as follows:

```
3 + 2 * 2 == 9
    \---/
3 +   4   == 9
\-----/
   7      == 9
   \---------/
      false
```

You can put arbitrary expressions on either side of the relational operator as long as they are of a compatible type. Here is a test with complex expressions on either side:

```
(2 - 3 * 8) / (435 % (7 * 2)) <= 3.8 - 4.5 / (2.2 * 3.8)
```

One limitation of these operators is that they should only be used with primitive data. Later in this chapter we will talk about how to compare objects for equality, and in a later chapter we'll discuss how to perform less-than and greater-than comparisons on objects.

Cumulative Sum with if

Let's now see how the use of if/else statements can allow us to create some interesting variations on the cumulative sum algorithm. Suppose that we want to read a sequence of numbers and compute the average. This seems like a straightforward variation of our cumulative sum code. We can compute the average as the sum divided by the number of numbers, as in:

```
System.out.println("average = " + sum / totalNumber);
```

There is one minor problem with this code. Suppose that when we ask the user how many numbers to process, the user says to process 0 numbers. That would mean that we never enter our cumulative sum loop and we try to compute the value of 0 divided by 0. Java would print out that the average is "NaN". This cryptic message is short for "Not a Number". It would be better to print out some other kind of message that would indicate that there weren't any numbers to average. We can use an if/else for this purpose:

```
if (totalNumber <= 0) {
    System.out.println("No numbers to average");
} else {
    System.out.println("average = " + sum / totalNumber);
}
```

We can fit in another use of if statements by counting how many negative numbers are entered by the user. You will often find yourself wanting to count how many times something occurs in a program. This goal is easy to accomplish with an if statement and an integer variable called a counter. You start by initializing the counter to 0:

```
int negatives = 0;
```

You can use any name you want for the variable. Here we used the name "negatives" because that is what we're counting. The other essential ingredient is to increment the counter inside the loop if it passes the test we're interested in:

```
if (next < 0) {
    negatives++;
}
```

Putting this all together and modifying the comments and introduction, we end up with the following variation of the cumulative sum program:

```
 1  // Finds the average of a sequence of numbers as well as reporting
 2  // how many of the numbers were negative.
 3
 4  import java.util.*;
 5
 6  public class ExamineNumbers2 {
 7      public static void main(String[] args) {
 8          System.out.println("This program examines a sequence of numbers to");
 9          System.out.println("find the average as well as counting how many");
10          System.out.println("are negative.");
11          System.out.println();
12
13          Scanner console = new Scanner(System.in);
14
15          System.out.print("How many numbers do you want me to examine? ");
16          int totalNumber = console.nextInt();
17
18          int negatives = 0;
19          double sum = 0.0;
20          for (int i = 1; i <= totalNumber; i++) {
21              System.out.print("    #" + i + "? ");
22              double next = console.nextDouble();
23              sum += next;
24              if (next < 0) {
25                  negatives++;
26              }
27          }
28          System.out.println();
29
30          if (totalNumber <= 0) {
31              System.out.println("No numbers to average");
32          } else {
33              System.out.println("average = " + sum / totalNumber);
34          }
35          System.out.println("# of negatives = " + negatives);
36      }
37  }
```

The program will execute something like this:

```
This program examines a sequence of numbers to
find the average as well as counting how many
are negative.

How many numbers do you want me to examine? 8
    #1? 2.5
    #2? 9.2
    #3? -19.4
    #4? 208.2
    #5? 42.3
    #6? 92.7
    #7? -17.4
    #8? 8

average = 40.7625
# of negatives = 2
```

Fencepost with if

Many of the fencepost loops that you write will require conditional execution. In fact, the fencepost problem itself can be solved with an if statement. Remember that the classic solution to the fencepost is to handle the first post before the loop begins:

```
plant a post.
for (the length of the fence) {
    attach some wire.
    plant a post.
}
```

This solution solves the problem but it can be confusing because inside the loop we do things in reverse order. With an if statement we can keep the original order of the steps:

```
for (the length of the fence) {
    plant a post.
    if (this isn't the last post) {
        attach some wire.
    }
}
```

This variation isn't used as often as the classic solution because it involves a double test. We have a loop test and we have a test inside the loop. Often these are nearly the same test, so it is inefficient to test the same thing twice each time through the loop. But there will be situations where you might use this approach. For example, in the classic approach, we repeat whatever lines of code correspond to planting a post. If there is a lot of code involved, we might decide that the if inside the loop is a better approach even if it leads to some extra testing.

As an example, let's consider writing a method called multiprint that will print a string a particular number of times. Suppose that we want the output on a line by itself inside square brackets and separated by commas. Below are two example calls.

```
multiprint("please", 4);
multiprint("beetlejuice", 3);
```

We expect these calls to produce the following output:

```
[please, please, please, please]
[beetlejuice, beetlejuice, beetlejuice]
```

If we don't think about the fencepost, we can write a simple loop that prints square brackets outside the loop and that prints the string with a comma inside the loop:

```
public static void multiprint(String s, int times) {
    System.out.print("[");
    for (int i = 1; i <= times; i++) {
        System.out.print(s + ", ");
    }
    System.out.println("]");
}
```

Unfortunately, this code produces an extraneous comma after the last value:

```
[please, please, please, please, ]
[beetlejuice, beetlejuice, beetlejuice, ]
```

Because the commas are separators, we want to print one more string than comma (e.g., two commas to separate the three occurrences of "beetlejuice"). We can use the classic solution to the fencepost problem to get this behavior by printing one string outside the loop and reversing the order of the printing inside the loop:

```
public static void multiprint(String s, int times) {
    System.out.print("[" + s);
    for (int i = 2; i <= times; i++) {
        System.out.print(", " + s);
    }
    System.out.println("]");
}
```

Notice that because we are printing one of the strings before the loop begins, we have to modify the loop so that it won't print as many strings as it did before. We have adjusted the loop variable i to start at 2 to account for the first value that is printed before the loop.

Unfortunately, this solution does not quite work properly. Think of what happens when we ask the method to print a string zero times, as in:

```
multiprint("please don't", 0);
```

This call produces the following incorrect output:

```
[please don't]
```

It should be possible to request zero occurrences of a string, so the method shouldn't do this. The problem is that the classic solution to the fencepost involves printing one value before the loop begins. To get it to behave correctly for the zero case, we would have to include an if/else:

```
public static void multiprint(String s, int times) {
    if (times == 0) {
        System.out.println("[]");
    } else {
        System.out.print("[" + s);
        for (int i = 2; i <= times; i++) {
            System.out.print(", " + s);
        }
        System.out.println("]");
    }
}
```

As an alternative, we can use the approach of including an if inside the loop (the double-test approach):

```
public static void multiprint(String s, int times) {
    System.out.print("[");
    for (int i = 1; i <= times; i++) {
        System.out.print(s);
        if (i < times) {
            System.out.print(", ");
        }
    }
    System.out.println("]");
}
```

Although this code performs a similar test twice on each iteration, it is simpler than the classic solution and its special case. Neither solution is better than the other. There is a tradeoff involved. If we think the code will be executed often with the loop iterating many times, we would be more inclined to use the efficient solution. Otherwise we might choose the simpler code instead.

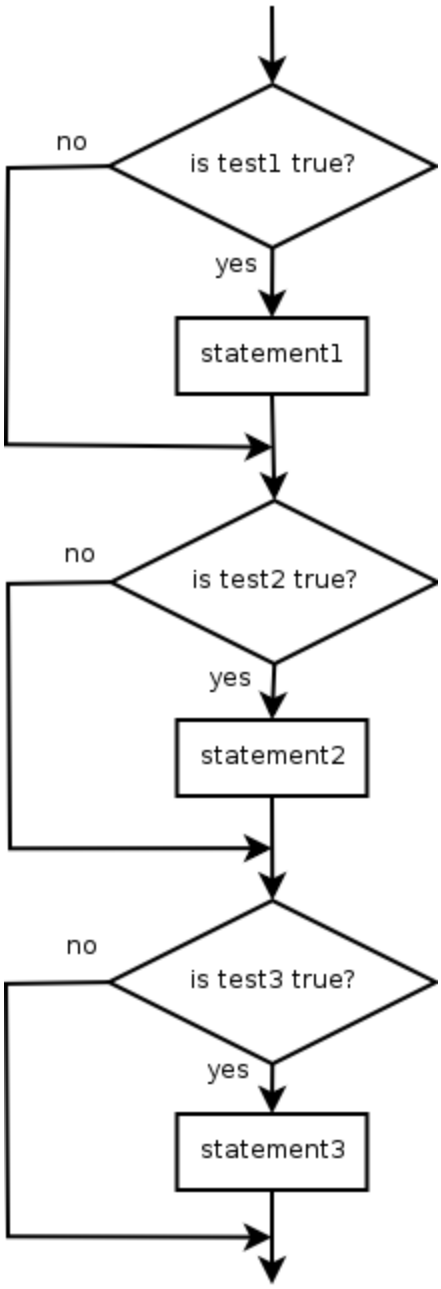
Nested if/else

Many beginners write code that looks like this:

```
if (<test1>) {
    <statement1>;
}
if (<test2>) {
    <statement2>;
}
if (<test3>) {
    <statement3>;
}
```

This sequential structure is appropriate if you want to execute any combination of the three statements. You might write this code in a program for a questionnaire with three optional parts, any combination of which might be applicable for a given person.

The following diagram shows the flow of the sequential if code. Notice that it's possible to execute none of the controlled statements (if all tests are false), or just one of them (if only that test happens to be true), or many of them (if multiple tests are true).



Often, however, you only want to execute one of a series of statements. In such cases, it is better to nest the if statements:

```
if (<test1>) {
    <statement1>;
} else {
    if (<test2>) {
        <statement2>;
    } else {
        if (<test3>) {
            <statement3>;
        }
    }
}
```

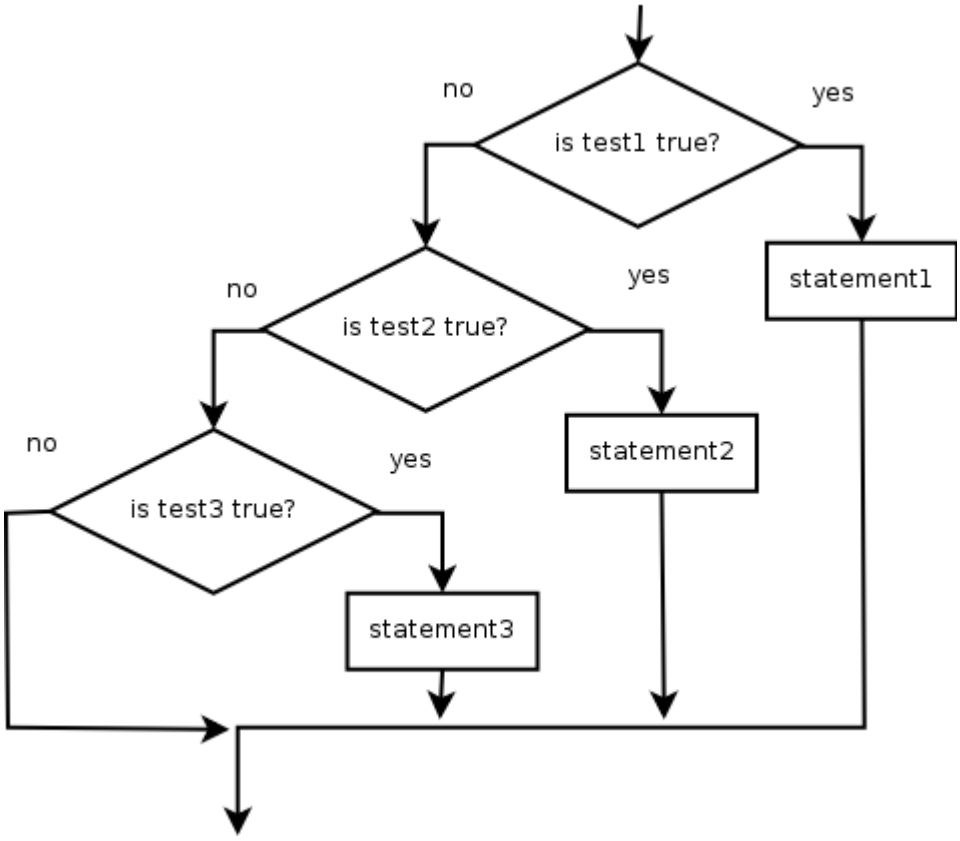
With this construct, you can be sure that one statement at most is executed. The first test to return true has its corresponding statement executed. If no tests return true, no statement is executed. If executing at most one statement is your objective, then this construct is more appropriate than the sequential if statements. It reduces the likelihood of errors and simplifies the testing process.

As you can see, nesting if statements like this leads to a lot of indentation. The indentation also isn't all that helpful because we really think of this as choosing one of a number of alternatives. K&R style has a solution for this as well. If an else is followed by an if, we put them on the same line:

```
if (<test1>) {
    <statement1>;
} else if (<test2>) {
    <statement2>;
} else if (<test3>) {
    <statement3>;
}
```

This way the various statements that we are choosing from all appear at the same level of indentation. Sun recommends that nested if/else statements be indented in this way.

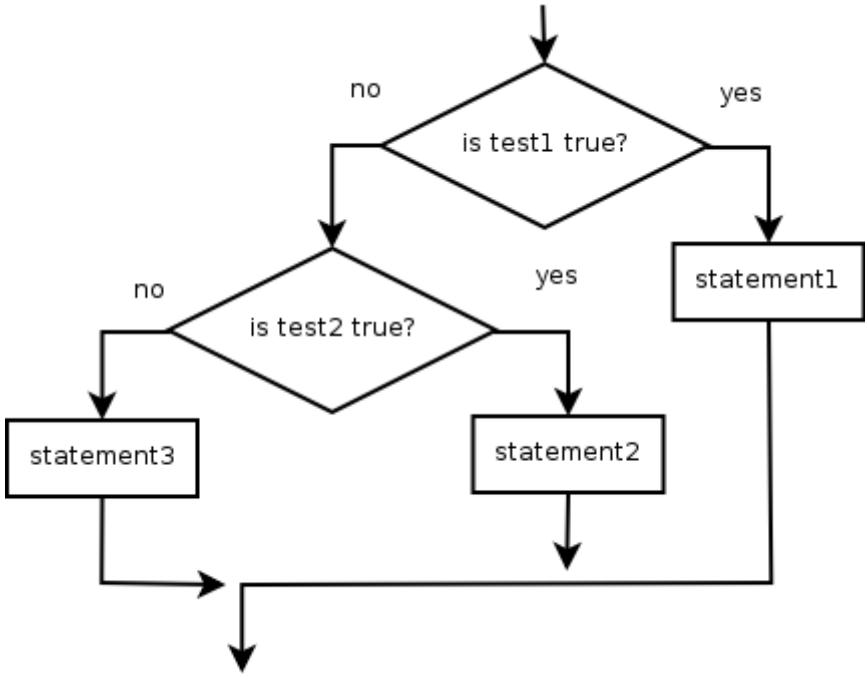
The following diagram shows the flow of the nested if/else code. Notice that it is possible to execute one of the controlled statements (the first one whose test is true), or none (if no tests are true).



There is a variation of this structure in which the final statement is controlled by an else instead of a test.

```
if (<test1>) {
    <statement1>;
} else if (<test2>) {
    <statement2>;
} else {
    <statement3>;
}
```

In this construct, the final branch will always be taken when all the tests fail and, thus, the construct will always execute exactly one of the three statements. The following diagram shows the flow of this modified nested if/else code.



To explore these variations, consider the task of writing out whether a number is positive, negative, or zero. You could structure this as three simple if statements as follows.

```
if (number > 0) {
    System.out.println("Number is positive.");
}
if (number == 0) {
    System.out.println("Number is zero.");
}
if (number < 0) {
    System.out.println("Number is negative.");
}
```

To determine how many of the printlns are potentially executed, you have to stop and think about the tests being performed. You shouldn't have to put that much effort into understanding this code. The code is clearer if we nest the if statements:

```
if (number > 0) {
    System.out.println("Number is positive.");
} else if (number == 0) {
    System.out.println("Number is zero.");
} else if (number < 0) {
    System.out.println("Number is negative.");
}
```

This solution, however, is not the best. You know that you want to execute one and only one println statement. This nested structure does not preclude the possibility of no statement being executed. If all three tests fail, no statement would be executed. With these particular tests that will never happen. If a number is neither positive nor zero, it must be negative. Thus, the final test here is unnecessary and misleading. You must think about the tests in order to know whether or not it is possible for all three of these branches to be skipped.

In this case, the best solution is the nested if/else with a final branch that is always taken if the first two tests fail:

```
if (number > 0) {
    System.out.println("Number is positive.");
} else if (number == 0) {
    System.out.println("Number is zero.");
} else {
    System.out.println("Number is negative.");
}
```

You can glance at this construct and see that exactly one println will be executed. You don't have to look at the tests being performed in order to realize this, it is a property of this kind of nested if/else. This is a good place to include a comment to make it clear what is going on:

```
if (number > 0) {
    System.out.println("Number is positive.");
} else if (number == 0) {
    System.out.println("Number is zero.");
} else {  // number must be negative
    System.out.println("Number is negative.");
}
```

One final benefit of this approach is efficiency. With three simple if statements, we always perform all three tests. With the nested if/else approach, we test only until we find a match. For example, in the code above we only need to perform one test for positive numbers and at most two tests overall.

When you find yourself writing code to pick among alternatives like these, you have to analyze the particular problem to figure out how many of the branches you potentially want to execute. If any combination can be taken, use sequential if statements. If you want one or none of the branches to be taken, use the nested if/else with a test for each statement. If you want exactly one branch to be taken, use the nested if/else with a final branch controlled by an else rather than by a test.

The table below summarizes these choices.

If/Else Options

Situation	Construct	Basic Form

You want to execute any combination of controlled statements	sequential ifs	`if (<test1>) {`
		`  <statement1>;`
		`}`
		`if (<test2>) {`
		`  <statement2>;`
		`}`
		`if (<test3>) {`
		`  <statement3>;`
		`}`

You want to execute 0 or 1 of the controlled statements	nested ifs ending in test	`if (<test1>) {`
		`  <statement1>;`
		`} else if (<test2>) {`
		`  <statement2>;`
		`} else if (<test3>) {`
		`  <statement3>;`
		`}`

You want to execute exactly 1 of the controlled statements	nested ifs ending in else	`if (<test1>) {`
		`  <statement1>;`
		`} else if (<test2>) {`
		`  <statement2>;`
		`} else {`
		`  <statement3>;`
		`}`

Common Programming Error: Choosing the Wrong if/else Construct

Suppose that your instructor has told you that grades will be determined as follows:

A for scores ≥ 90
B for scores ≥ 80
C for scores ≥ 70
D for scores ≥ 60
F for scores < 60

We can translate this into code as follows:

```
String grade;
if (score >= 90) {
    grade = "A";
}
if (score >= 80) {
    grade = "B";
}
if (score >= 70) {
    grade = "C";
}
if (score >= 60) {
    grade = "D";
}
if (score < 60) {
    grade = "F";
}
```

The first problem we encounter is that if we try to use the variable grade after this code, we get this error from the compiler:

```
variable grade might not have been initialized
```

This should already be a clue that we have a problem. The Java compiler is saying that there are paths through this code that would leave the variable grade uninitialized. We can fix this by giving an initial value to grade:

```
String grade = "no grade";
```

This change allows the code to compile. But if you compile and run it, you will find that it gives out only two grades: D and F. Anyone who has a score of at least 60 ends up with a D and anyone with a grade below 60 ends up with an F. And even though the compiler complained that there was a path that would allow grade not to be initialized, we never end up with someone getting a grade of "no grade".

The problem here is that we want to execute exactly one of these assignment statements. But with the sequential if statements, we can execute several of them. For example, if someone has a score of 95, we set grade to "A", then reset it to "B", then reset it to "C" and finally reset it to "D". We can fix this by using a nested if/else construct:

```
String grade;
if (score >= 90) {
    grade = "A";
} else if (score >= 80) {
    grade = "B";
} else if (score >= 70) {
    grade = "C";
} else if (score >= 60) {
    grade = "D";
} else {  // score < 60
    grade = "F";
}
```

We don't even need to set grade to "no grade" because the compiler can see that no matter what path we follow, the variable grade will be assigned a value (exactly one of the branches will be executed).

4.3 Subtleties of Conditional Execution

The last section presented the basic idea behind if statement and if/else constructs. You'll find as we get deeper into programming that many of the new constructs we study aren't overly complex, but they take time to master. This is true of any craft. For example, it would probably take only a day to learn each of the major tools used by a carpenter, but learning when to use which tool, how to use them together and how to use them well would require much more than a day of training.

In this section we will explore some of the issues that arise when you start using conditional execution.

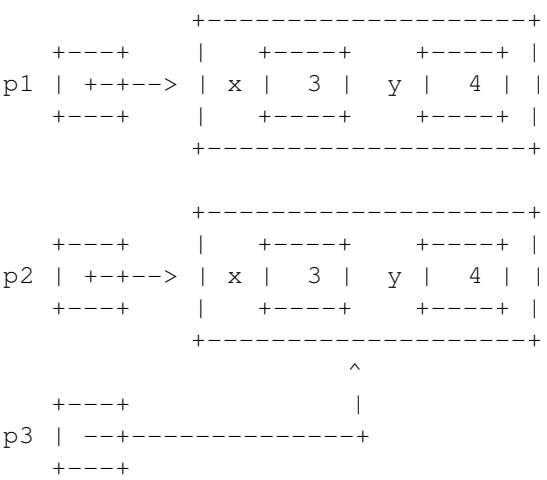
Object Equality

We saw earlier in the chapter that you can use the == and != operators to test for equality and non-equality, respectively. Unfortunately, these operators do not work as you might expect when testing for equality and non-equality of objects like Strings and Points, so we will have to learn a new way to test objects for equality.

Suppose that we were to execute the following lines of code.

```
Point p1 = new Point(3, 4);
Point p2 = new Point(3, 4);
Point p3 = p2;
```

It will lead to three variables and two objects:



Even though we have two Point objects, they both have coordinates (3, 4). So we would normally think of these points as being equal. But if we were to test whether (p1 == p2) the answer would be false. In performing this test, Java sees whether the variables p1 and p2 store exactly the same value. These variables store references to Point objects, but they aren't the Point objects themselves. So this test is asking whether p1 and p2 refer to the same object. Since they refer to different objects, the "==" test evaluates to false.

If we were instead to ask whether (p2 == p3) the answer would be true. That's because p2 and p3 store the same value. In other words, they both refer to the same object.

Java provides a second way of testing for equality that is intended for testing object equality. Every Java object has a method called "equals" that takes another object as an argument. So you ask an object whether it equals another object. For example, given the variables above we could test whether:

```
if (p1.equals(p2)) {
    System.out.println("points are equal");
}
```

In this case we are not comparing the variables, we are calling a method of the Point object referred to by p1. For Point objects, the equals method compares the x and y coordinates stored in each point to see if they are the same. In effect, we are saying, "Hey, p1, do you store the same x and y coordinates as p2?" In this case, the test evaluates to true.

You will want to use the equals method to test for equality of Strings. For example, you might write code like the following to read a token from the console and to call one of two different methods depending upon whether the user responded with "yes" or "no". If the user types neither word, it prints an error message.

```
System.out.print("yes or no? ");
String s = console.next();
if (s.equals("yes")) {
    processYes();
} else if (s.equals("no")) {
    processNo();
} else {
    System.out.println("You didn't type yes or no");
}
```

This code will not work as intended with "==" tests.

Java has a special variation of the equals method for the String class that ignores case differences (capital versus small letters). The method is called equalsIgnoreCase. For example, you could rewrite the code above to recognize responses like "Yes", "YES", "No", NO", yES", etc.

```
System.out.print("yes or no? ");
String s = console.next();
if (s.equalsIgnoreCase("yes")) {
    processYes();
} else if (s.equalsIgnoreCase("no")) {
    processNo();
} else {
    System.out.println("You didn't type yes or no");
}
```

Roundoff Errors

Earlier in the chapter we looked at a program called ExamineNumbers2 that will read a series of numbers and compute the average of the numbers and that will count how many negatives are in the sequence. Below is a sample execution that produces an unusual final result:

```
This program examines a sequence of numbers to
find the average as well as counting how many
are negative.

How many numbers do you want me to examine? 4
    #1? 2.1
    #2? -3.8
    #3? 5.4
    #4? 7.4

average = 2.7750000000000004
# of negatives = 1
```

If you use a calculator, you will find that the four numbers add up to 11.1. If you divide 11.1 by 4 in your calculator, you get 2.775. Yet Java is reporting this as 2.7750000000000004. Where do all of those zeros come from and why does it end in 4? The answer is that floating point numbers can lead to *round off errors*.

Round Off Error

A numerical error that occurs because floating point numbers are stored as approximations rather than as exact values.

Round off errors are generally small and can occur in either direction (slightly high or slightly low). In the case above, we got a roundoff error that was slightly high.

It shouldn't be too difficult to understand why this occurs. Remember that floating point numbers are stored in a format similar to scientific notation with a set of digits and an exponent. Think of how you would store one-third in scientific notation using base 10. We describe it as 3.33333 (repeating) times 10 to the -1 power. We can't store an infinite number of digits on a computer, so we would have to stop repeating the 3's at some point. Suppose we could store 10 digits. Then we'd store it as 3.333333333 times 10 to the -1. If we multiply that by 3, we don't get back 1. Instead, we get 9.999999999 times 10 to the -1 (which is equal to 0.9999999999).

You might wonder why the numbers we used would cause a problem when they didn't have any repeating digits. You have to remember that the computer stores numbers in base 2. Numbers like 2.1 and 5.4 might look like simple numbers in base 10, but they have repeating digits when stored in base 2.

Roundoff errors can lead to rather surprising outcomes. For example, consider the following short program:

```
1  public class Roundoff {
2      public static void main(String[] args) {
3          double n = 1.0;
4          for (int i = 1; i <= 10; i++) {
5              n += 0.1;
6              System.out.println(n);
7          }
8      }
9  }
```

This is a classic cumulative sum where we add 0.1 to the number each time through the loop. We start with n equal to 1.0 and the loop iterates ten times, so this should print the numbers 1.1, 1.2, 1.3, and so on through 2.0. Here is the output it produces

```
1.1
1.2000000000000002
1.3000000000000003
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
2.000000000000001
```

We get unusual output because 0.1 cannot be stored exactly in base 2 (it produces a repeating set of digits just as one third does in base 10). Each time through the loop, we compound the error, which is why the roundoff error gets worse each time.

As another example, consider the task of adding up a penny, a nickel, a dime and a quarter. If we use variables of type int, we will get an exact answer no matter what order we add the numbers:

```
int cents1 = 1 + 5 + 10 + 25;
int cents2 = 25 + 10 + 5 + 1;
System.out.println(cents1);
System.out.println(cents2);
```

The output of this code is as follows:

```
41
41
```

No matter what order we use to add the numbers, we'll find that it adds up to 41 cents. But suppose that instead of thinking of these as cents, we think of them as fractions of a dollar that we would store with doubles:

```
double dollars1 = 0.01 + 0.05 + 0.10 + 0.25;
double dollars2 = 0.25 + 0.10 + 0.05 + 0.01;
System.out.println(dollars1);
System.out.println(dollars2);
```

This code has a surprising output:

```
0.41000000000000003
0.41
```

Even though we are adding up exactly the same numbers, the fact that we add them in a different order makes a difference. The reason is roundoff errors.

There are several lessons to draw from this:

- Realize that when you store values with a floating point type like double, you are storing approximations and not exact values. If you need something stored exactly, store it using type int which has an exact representation.
- Don't be surprised when you see numbers that are slightly off from the expected value.
- Don't expect to be able to compare variables of type double for equality.

To follow up on the third point, consider what would happen with the code above if we were to perform the following test:

```
if (dollars1 == dollars2) {
    ...
}
```

The test would evaluate to false because they are very close, but not close enough for Java to consider them equal. We rarely use a test for exact equality when we work with doubles. Instead, we can test to see if numbers are close to each other using a test like this:

```
if (Math.abs(dollars1 - dollars2) < 0.001) {
    ...
}
```

We use the absolute value method from the Math class to find the magnitude of the difference and see whether it is less than some small epsilon (in this case, 0.001).

Later in the chapter we will see a variation on print/println called printf that will make it easier to print numbers like these without all of the extra digits.

Factoring if/else Statements

Suppose you are writing a program that plays a betting game with a user and you want to give different warnings about how much cash the user has. The following nested if/else distinguishes three different cases: money less than $500, which is considered low; money between $500 and $1000, which is considered okay; and money over $1000, which is considered good. Notice that the user is given different advice in each branch:

```
if (money < 500) {
    System.out.println("You have, $" + money + " left.");
    System.out.print("Your cash is dangerously low.  Bet carefully.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
} else if (money < 1000) {
    System.out.println("You have, $" + money + " left.");
    System.out.print("Your cash is somewhat low.  Bet moderately.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
} else {
    System.out.println("You have, $" + money + " left.");
    System.out.print("Your cash is in good shape.  Bet liberally.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
}
```

This construct is repetitious and can be reduced using a technique called factoring. With it, you factor out common pieces of code from the different branches of the if/else. The technique is simple. The above construct creates three different branches depending upon the value of the variable money. You start by writing down the series of actions being performed in each branch and comparing them.

```
                    Depending upon money:
                                |
        +-----------------------+-----------------------+
        |                       |                       |
   money < 500          500 <= money < 1000        Money >= 1000
        |                       |                       |
Report money             Report money             Report money
Warn about low money     Report okay money        Report good money
Prompt for bet           Prompt for bet           Prompt for bet
Read bet                 Read bet                 Read bet
```

You can factor at both the top and the bottom of such a construct. If you notice that the top statement in each branch is the same, you factor it out of the branching part and put it before the branch. Similarly, if the bottom statement in each branch is the same, you factor it out of the branching part and put it after the loop. You can factor the top statement in each of these branches and the bottom two statements.

```
                          Report money

                    Depending upon money:
                                |
        +-----------------------+-----------------------+
        |                       |                       |
   money < 500          500 <= money < 1000        Money >= 1000
        |                       |                       |
Warn about low money     Report okay money        Report good money

                          Prompt for bet
                            Read bet
```

Thus, the code above can be reduced to the following which is more succinct.

```
System.out.println("You have, $" + money + " left.");
if (money < 500) {
    System.out.print("Your cash is dangerously low.  Bet carefully.");
} else if (money < 1000) {
    System.out.print("Your cash is somewhat low.  Bet moderately.");
} else {
    System.out.print("Your cash is in good shape.  Bet liberally.");
}
System.out.print("How much do you want to bet? ");
bet = console.nextInt();
```

Min/Max Loops

We have seen how to compute the average of a sequence of numbers. Another common programming task is to keep track of the maximum and/or minimum value in a sequence. For example, with an average temperature of around 50 degrees below zero Fahrenheit, the surface of the Moon is clearly inhospitable. But the much greater challenge of living on the Moon is that the range of temperatures is from around 240 degrees below zero to around 250 degrees above zero.

To compute the maximum of a sequence of values, we can keep track of the largest value we've seen so far and use an if statement to update the maximum if we come across a new value that is larger than the current maximum. We can describe this approach in pseudocode as follows:

```
initialize max.
for (all numbers to examine) {
    obtain "next".
    if (next > max) {
        max = next;
    }
}
```

Initializing the maximum isn't quite as simple as it sounds. For example, novices often initialize max to 0. But what if the sequence of numbers you are examining is composed entirely of negative numbers? For example, you might be asked to find the maximum of this sequence:

```
-84, -7, -14, -39, -410, -17, -41, -9
```

The correct answer is that -7 is the maximum value in this sequence. But you would never set max to -7 because 0 is larger. So you would incorrectly report the maximum as 0.

There are two classic solutions to this problem. First, if you know the range of the numbers you are examining, then you have an appropriate choice for max. In that case, you can set max to the lowest value in the range. That seems counterintuitive because normally we think of the max as being large, but the idea is to set the max to the smallest possible value it could ever be so that anything larger will cause us to reset the max to that value. For example, if the numbers above are temperatures in degrees Fahrenheit, we know that they will never be smaller than absolute zero (around -460 degrees Fahrenheit). So we could initialize max to that value.

The second possibility is to initialize max to the first value in the sequence. That won't always be convenient because it means obtaining one of the numbers outside the loop.

Combining these two possibilities, our pseudocode becomes:

```
initialize max either to lowest possible value or to first value.
for (all numbers to examine) {
    obtain "next".
    if (next > max) {
        max = next;
    }
}
```

The pseudocode for computing the minimum is a slight variation of this:

```
initialize min either to highest possible value or to first value.
for (all numbers to examine) {
    obtain "next".
    if (next < min) {
        min = next;
    }
}
```

To understand this better, let's put the pseudocode into action with a real problem. In Mathematics there is an open problem that involves what are known as *hailstone sequences*. They have that name because they have the property that they often rise and fall in unpredictable patterns which is somewhat analogous to the process that forms hailstones.

A hailstone sequence is a sequence of numbers in which each value x is followed either by:

(3x + 1) if x is odd
(x/2) if x is even

For example, if we start with 7 and we construct a sequence of length 10, we get:

7, 22, 11, 34, 17, 52, 26, 13, 40, 20

In this sequence, the maximum and minimum values are 52 and 7, respectively. If we extend this to a sequence of length 20, we get:

7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1

In this case, the maximum and minimum values are 52 and 1, respectively.

You will notice that once you get to any of the numbers 1, 2 or 4, the sequence repeats itself. It is conjectured that all integers eventually reach 1, like hailstones that fall to the ground. This is an unsolved problem in Mathematics. Nobody has been able to disprove it, but nobody has proven it either.

Let's write a method that takes a starting value and a sequence length and that prints the maximum and minimum values obtained in a hailstone sequence formed with that starting value and of that length. So our method will look like this:

```
public static void printHailstoneMaxMin(int value, int length) {
    ...
}
```

We can use the starting value to initialize max and min:

```
int min = value;
int max = value;
```

We then need a loop that will generate the other values. We are passed a parameter telling us how many times to go through the loop, but we don't want to execute length times. Remember that the starting value is part of the sequence. So if we want to use a sequence of the given length, we have to make sure that the number of iterations is one less than length. Combining this idea with the max/min pseudocode, we know the loop will look like this:

```
for (int i = 1; i <= length - 1; i++) {
    compute next number.
    if (value > max) {
        max = value;
    }
    if (value < min) {
        min = value;
    }
}
print max and min.
```

To fill out the pseudocode for "compute next number", we need to translate the hailstone formula into code. The formula is different depending upon whether the current value is odd or even. We can use an if/else to solve this task. For the test, we can use a "mod 2" test to see what remainder we get when we divide by 2. Even numbers have a remainder of 0 and odd numbers have a remainder of 1. So the test should look like this:

```
if (value % 2 == 0) {
    do even computation.
} else {
    do odd computation.
}
```

Translating the hailstone mathematical formulas into Java expressions, we get:

```
if (value % 2 == 0) {
    value = value / 2;
} else {
    value = 3 * value + 1;
}
```

The only part of our pseudocode that we haven't filled in yet is the printing part after the loop. That is fairly easy to complete, which gives us the following complete method:

```
public static void printHailstoneMaxMin(int value, int length) {
    int min = value;
    int max = value;
    for (int i = 1; i <= length - 1; i++) {
        if (value % 2 == 0) {
            value = value / 2;
        } else {
            value = 3 * value + 1;
        }
        if (value > max) {
            max = value;
        }
        if (value < min) {
            min = value;
        }
    }
    System.out.println("max = " + max);
    System.out.println("min = " + min);
}
```

4.4 Text Processing

Programmers commonly face problems that require them to create, edit, examine, and format text. Collectively we call these types of tasks *text processing*.

Text processing

Editing and formatting strings of text.

In this section, we'll look at a new primitive type named `char` and a new command named System.out.printf, both of which are very useful for text processing tasks.

The char Type

In the previous chapter we learned about String objects that consist of zero to many characters. There is a primitive type named `char` that represents a single character of text. It's legal to have variables, parameters, and return values of type `char` if you so desire. Literal values of type `char` are expressed by placing the character within single quotes, such as:

```
char ch = 'A';
```

It is also legal to create a char value that represents an escape sequence, such as:

```
char newline = '\n';
```

The distinction between char and String is a subtle one that confuses many new Java programmers. Why have two types for such similar data? Type char exists primarily for historical reasons dating back to older languages such as C that influenced the design of Java.

So why would a person ever use type char when String is available? In many cases we'll use char out of necessity, because some methods in Java's API use type char as a parameter or return type. But there are a few cases where char can be more useful or simpler than String.

Values of type char can be used in simple arithmetic expressions and in loops to cover ranges of letters. The char values can also be compared using relational operators such as < or ==. Strings don't have either of these abilities. For example, the following code prints every letter of the alphabet:

```
// prints abcdefghijklmnopqrstuvwxyz
for (char letter = 'a'; letter <= 'z'; letter++) {
    System.out.print(letter);
}
```

The main bridge between types String and char is the String object's `charAt` method, which accepts an integer index as a parameter and returns the String's character at that index. We often break apart Strings based on characters, or do loops over Strings that examine or change each character. For example, the following loop removes any occurrences of the space character `' '` from a String:

```
String phrase = "the rain in Spain";
String noSpaces = "";
for (int i = 0; i < phrase.length(); i++) {
    char ch = phrase.charAt(i);
    if (ch != ' ') {
        noSpaces += ch;
    }
}
// noSpaces stores "therainin Spain" here
```

This example uses an interesting technique in that it builds a String using a loop, starting with an empty String and concatenating individual characters onto it. This is sometimes called a *cumulative concatenation*, which is a variation of the cumulative sum technique shown earlier in the chapter. At the end of the preceding code, the variable noSpaces stores the String `"therainInSpain"`. The code also illustrates that char values can be concatenated onto a String like other primitive values can.

There are several useful methods that can be called to check information about a character or convert one character into another. Remember that char is a primitive type, which means that you can't use the `objectName.methodName()` syntax like we did with Strings. Instead, the methods are static methods in the Character class, which accept char parameters and return appropriate values. Here are some of the most useful Character methods:

Useful Methods of the Character Class

Method	Description	Example
`getNumericValue(char)`	converts a character that looks like a number into that number	`Character.getNumericValue('6')` returns `6`
`isDigit(char)`	whether or not character is '0' through '9'	`Character.isDigit('X')` returns `false`
`isLetter(char)`	whether or not character is in range 'a' to 'z' or 'A' to 'Z'	`Character.isLetter('f')` returns `true`
`isLowerCase(char)`	whether or not character is a lowercase letter	`Character.isLowerCase('Q')` returns `false`
`isUpperCase(char)`	whether or not character is an uppercase letter	`Character.isUpperCase('Q')` returns `true`
`toLowerCase(char)`	the lowercase version of the given letter	`Character.toLowerCase('Q')` returns `'q'`
`toUpperCase(char)`	the uppercase version of the given letter	`Character.toUpperCase('x')` returns `'X'`

For example, the following program counts the number of letters in a String `s`, ignoring all non-letter characters.

```
1  // Counts letter characters in a String.
2  import java.util.*;   // for Scanner
3
4  public class CountLetters {
5      public static void main(String[] args) {
6          Scanner console = new Scanner(System.in);
7          System.out.print("Type a word: ");
8          String s = console.next();
9
10         int count = 0;
11         for (int i = 0; i < s.length(); i++) {
12             char ch = s.charAt(i);
13             if (Character.isLetter(ch)) {
14                 count++;
15             }
16         }
17         System.out.println("Letters: " + count);
18     }
19 }
```

The program produces the following output for an input of "test123TEST":

```
Type a word: test123TEST
Letters: 8
```

Every character also has a corresponding integer value. Java is willing to automatically convert a value of type char into an int whenever it is expecting an int. For example, you can form the following rather strange expression:

```
2 * 'a'
```

Java sees a value of type char where it was expecting an int, but it's happy to convert this for you. It turns out that the integer value for 'a' is 97, so the expression's result is 194.

Java will not, however, automatically convert in the other direction. That is because the type char is in a sense a subset of the type int. Every char value has a corresponding int, but not every int has a corresponding char. That means that Java would always be able to turn a char into an int, but might not always be able to turn an int into a char. So in this case Java insists on a cast. This allows you to form expressions like the following:

```
(char) ('a' + 3)
```

Inside the parentheses we have a combination of a char value and an int value, so Java converts the char to an int (97) and then performs the addition operation, yielding the value 100. This integer is then converted to a char value because of the cast, which yields the character 'd' (the character that appears 3 later in the sequence than the character 'a').

Did you Know: ASCII and Unicode

We store data on a computer as binary numbers (sequences of 0s and 1s). To store textual data, we need an encoding scheme that will tell us what sequence of 0s and 1s to use for any given character. Think of it as being like a giant secret decoder ring that says things like, "If you want to store a little 'a', then use the sequence 01100001."

In the early 1960's IBM developed an encoding scheme called *EBCDIC* that worked well with their punched cards. Punched cards were in use for decades before computers were even invented. But it soon became clear that EBCDIC wasn't a convenient encoding scheme for computer programmers to work with. For example, there were gaps in the sequence where characters like 'i' and 'j' were far apart even though we think of them as coming one after the other.

In 1967 the American Standards Association published a scheme known as *ASCII* (pronounced "AS-kee") that has been in common use ever since. The acronym is short for "American Standard Code for Information Interchange." In its original form, ASCII defined 128 characters that could be stored with 7 bits of data.

The biggest problem with ASCII is that it is an *American* code. There are many characters in common use in other countries that were not included in ASCII. For example, the British pound (£) and the Spanish variant of the letter n (ñ) are not included in the standard 128 ASCII characters. Various attempts have been made to add extra characters to ASCII to double it to 256 characters that include many of these special characters, but it turns out that 256 characters is simply not enough to capture the incredible diversity of human communication.

Around the time that Java was created a consortium of software professionals introduced a new standard for encoding characters known as *Unicode*. They decided that the 7 bits of standard ASCII and the 8 bits of extended ASCII were simply not big enough. They have not set a limit on how many bits they might use for encoding characters. As of the time of writing, the consortium has identified over 90 thousand characters, which requires a little over 16 bits to store. It includes the characters used in most modern languages and even some ancient languages. It does not yet include Egyptian or Mayan hieroglyphs and a proposal to include Klingon characters was rejected by the consortium.

The designers of Java decided to use Unicode as the standard for the type char, which means that Java programs are capable of manipulating a full range of characters. Fortunately the Unicode Consortium decided to incorporate the ASCII encodings, so ASCII can be seen as a subset of Unicode. If you are curious about the actual ordering of characters in ASCII, type "ASCII table" into your favorite search engine and you will find millions of hits to explore.

System.out.printf

So far we've seen System.out.println and System.out.print for console output. There's a third method named System.out.printf that is a bit more complicated than the others but gives us some useful new abilities. The 'f' in printf stands for "formatted," implying that System.out.printf gives you more control over the format in which your output is printed.

Imagine that you'd like to print a multiplication table from 1*1 up to 10*10. The following code prints the correct numbers, but it doesn't look very nice:

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.print(i * j + " ");
    }
    System.out.println();
}
```

The output is the following. Notice that the numbers don't line up horizontally.

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Separating the numbers by tabs is better, but the numbers are left-aligned, like in the shortened table below. What if we want them right-aligned?

```
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
4       8       12      16      20
5       10      15      20      25
```

It would be a pain to try to right-align the numbers, because we'd have to use if/else statements to check whether a given number was in a certain range, and pad it with a given number of spaces. A much easier way to print values aligned in fixed-width fields is the System.out.printf command. The printf method accepts a specially written String called a *format String* specifying the general appearance of the output, followed by any parameters to be printed.

```
System.out.printf(<format String>, <parameters>);
```

A format String is like a normal String except that it can contain placeholders called *format specifiers* where you specify a location for insertion of a variable's value, along with the format you'd like to give that value. Format specifiers begin with a % sign, and end with a letter specifying the kind of value, such as `d` for decimal integers (int) or `f` for floating-point real numbers (double). For example, the following code prints an x and y coordinate surrounded by parentheses:

```
int x = 38,  y = -152;
System.out.printf("%d %d", x, y);                // 38 -152
System.out.printf("location: (%d, %d)", x, y);   // location: (38, -152)
```

The `%d` is not actually printed out but is instead replaced with the corresponding parameter written after the format String. The number of format specifiers in the format String must match the number of parameters that follow it. The first specifier will be replaced by the first parameter, the second specifier by the second parameter, and so on. System.out.printf is a bizarre method in this way, because it can accept a varying number of parameters.

The printf command is like System.out.print in that it doesn't move to a new line unless you explicitly tell it to do so. A better format String above might have been "location: (%d, %d)\n" because it would complete the line.

Format specifiers can contain information between the % and the letter for the type, to specify the width, precision, and alignment of the value. For example, `%8d` specifies an integer right-aligned in an 8-space-wide area, and `%12.4f` specifies a double value right aligned in a 12-space-wide area rounded to 4 digits past the decimal point.

The following table lists some common format specifiers you may wish to use in your programs:

Specifier	Result
`%d`	integer
`%8d`	integer, right-aligned, 8 space wide field
`%-6d`	integer, left-aligned, 6 space wide field
`%f`	real number
`%12f`	real number, right-aligned, 12 space wide field
`%.2f`	real number, rounded to nearest hundredth
`%16.3f`	real number, rounded to nearest thousandth, 16 space wide field
`%s`	string
`%8s`	string, right-aligned, 8 space wide field
`%-9s`	string, left-aligned, 9 space wide field

As a comprehensive example, consider that the following variables have been declared.

```
int x = 38,  y = -152;
int grade = 86;
double angle = 87.4163;
String veggie = "carrot";
```

The following code sample prints the preceding variables with several format specifiers:

```
System.out.printf("hello there\n");
System.out.printf("x=%d and y=%d\n", x, y);
System.out.printf("%d%%\n", (grade + 5));
System.out.printf("!%d!%6d%6d\n", grade, x, y);
System.out.printf("%.2f %10.1f\n", angle);
System.out.printf("%s%12s!%-8s!\n", veggie, veggie, veggie);
```

The code produces the following output:

```
hello there
x=38 and y=-152
91%
!86!    38  -152
87.42       87.4
carrot      carrot!carrot  !
```

Let's return to our multiplication table example. Now that we know about printf, we can print the table with right-aligned numbers relatively easily. We'll right-align the numbers into fields of width 5:

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.printf("%5d", i * j);
    }
    System.out.println();
}
```

The code produces the following output:

```
    1    2    3    4    5    6    7    8    9   10
    2    4    6    8   10   12   14   16   18   20
    3    6    9   12   15   18   21   24   27   30
    4    8   12   16   20   24   28   32   36   40
    5   10   15   20   25   30   35   40   45   50
    6   12   18   24   30   36   42   48   54   60
    7   14   21   28   35   42   49   56   63   70
    8   16   24   32   40   48   56   64   72   80
    9   18   27   36   45   54   63   72   81   90
   10   20   30   40   50   60   70   80   90  100
```

In fact, it's also relatively simple to do a division table, should you ever need one. We'll use double values and specify a limited number of digits (3) after the decimal point.

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.printf("%8.3f", (double) i / j);
    }
    System.out.println();
}
```

The code produces the following output:

```
   1.000   0.500   0.333   0.250   0.200
   2.000   1.000   0.667   0.500   0.400
   3.000   1.500   1.000   0.750   0.600
   4.000   2.000   1.333   1.000   0.800
   5.000   2.500   1.667   1.250   1.000
```

As a final example, printf can elegantly solve the problem with the Roundoff program introduced in a previous section. By fixing the precision of the double value, it will be rounded to avoid the tiny roundoff mistakes that result from double arithmetic. Here is the fixed program:

```
1  public class Roundoff2 {
2      public static void main(String[] args) {
3          double n = 1.0;
4          for (int i = 1; i <= 10; i++) {
5              n += 0.1;
6              System.out.printf("%3.1f\n", n);
7          }
8      }
9  }
```

The program produces the following output:

```
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2.0
```

4.5 Methods with Conditional Execution

We learned a great deal about methods in Chapter 3. We learned about how to use parameters to pass values into a method and how to use a return statement to have a method return a value. Now that we have seen conditional execution, we need to revisit these issues so that we can gain a deeper understanding of them.

Preconditions and Postconditions

Every time you write a method you should think about exactly what that method is supposed to accomplish. You can describe how a method works by describing the *preconditions* that must be true before it executes and the *postconditions* that will be true after it is done executing.

Precondition

A condition that must be true before a method executes to guarantee that it can perform its task.

Postcondition

A condition that the method guarantees will be true after it finishes executing as long as the preconditions were true before it was called.

For example, to describe the job of a person on an auto assembly line, you might use a postcondition like, "The bolts that secure the left front tire are on the car and tight." But postconditions are not the whole story. Employees on an assembly line depend on each other. A line worker can't add bolts and tighten them if the left tire isn't there or if there are no bolts. So the assembly line worker might have preconditions like, "the left tire is mounted properly on the car; there are at least 8 bolts in the supply box; and a working wrench is available." You describe the task fully, then, by saying that the worker can make the postconditions true if the preconditions are true before starting.

Methods, like workers on an assembly line, need to work together, each solving its portion of the task, in order to solve the overall task. The preconditions and postconditions describe the dependencies between methods. To understand this better, consider this nonprogramming example with the following methods and their conditions.

```
makeBatter
        pre : bowl is clean.
        post: bowl has batter in it; bowl is dirty.
bakeCake
        pre : bowl has batter in it; pan is clean.
        post: cake is baked; pan is dirty.
washDishes
        pre : none.
        post: bowl and pan are clean.
```

When you call these methods, you must fit them together so the preconditions are always satisfied before a method executes. For example, if you suppose that the bowl and pan are initially clean, you can make a cake as follows.

```
makeBatter.
bakeCake.
```

Here is a trace of its execution.

```
--> bowl and pan are clean.
makeBatter.
--> bowl has batter in it; bowl is dirty; pan is clean.
bakeCake.
--> cake is baked; bowl and pan are dirty.
```

The preconditions are satisfied before each method is executed. However, if you want a loop to make cakes, you can't do it this way.

```
for (many cakes) {
    makeBatter.
    bakeCake.
}
```

The first cake is made properly, but not the second. The error occurs because the preconditions for makeBatter are not satisfied on the second execution of the loop.

You need a clean bowl and pan to execute makeBatter.

```
--> bowl and pan are clean.
makeBatter.
--> bowl has batter in it; bowl is dirty; pan is clean.
bakeCake.
--> cake is baked; bowl and pan are dirty.
makeBatter.
```

You need to change your solution.

```
for (many cakes) {
    makeBatter.
    bakeCake.
    washDishes.
}
```

The execution of washDishes leaves you with a clean bowl and pan and guarantees that you satisfy the preconditions of makeBatter on the next iteration.

Throwing Exceptions

We have seen several cases where Java might throw an exception. For example, if we have a console Scanner and we call nextInt, an exception will be thrown if the user types something that isn't an int. In Chapter 6 we will see how you can handle exceptions. For now we just want to explore some of the ways in which exceptions can occur and how we might want to generate them in our own code.

Ideally programs execute without generating any errors, but in practice various problems arise. You might ask the user for an integer and the user will accidentally or perhaps even maliciously type something that is not an integer. Or you might execute code that has a bug in it.

Below is a program that always throws an exception because it tries to compute the value of 1 divided by 0, which is mathematically undefined.

```
1  public class CauseException {
2      public static void main(String[] args) {
3          int x = 1 / 0;
4          System.out.println(x);
5      }
6  }
```

When you run the program, you get the following error message:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at CauseException.main(CauseException.java:3)
```

The problem is in line 3 when you ask Java to compute a value that can't be stored as an int. What is Java supposed to do with that value? It could ignore the error and continue executing, but that may not be the right thing to do. Instead Java throws an exception that stops the program from executing and gives us a warning that an arithmetic exception occurred while executing this line of code.

It is worth noting that division by 0 does not always produce an exception. We won't get an exception if we execute this line of code instead:

```
double x = 1.0 / 0.0;
```

The program executes normally and produces the output "Infinity". This is because floating-point numbers follow a standard from the IEEE (the Institute of Electrical and Electronics Engineers) that defines exactly what should happen in these cases and there are special values representing infinity and something called "NaN" (Not a Number).

We may want to throw exceptions ourselves in the code we write. In particular, it is a good idea to throw an exception if a precondition fails. For example, suppose that we want to write a method for computing the factorial of an integer. The factorial is defined as follows.

```
n! (which is read as "n factorial") = 1 * 2 * 3 * ... * n
```

We can write a Java method that uses a cumulative product to compute this result:

```
public static int factorial(int n) {
    int product = 1;
    for (int i = 2; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

We could test the method for various values with a loop:

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

which produces the following output:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

It seems odd that the factorial method should return 1 when you ask for 0!, but that is actually part of the mathematical definition of the factorial function. It is returning 1 because the local variable product in the factorial method is initialized to 1 and we never enter the loop when the parameter n has the value 0. So this is actually desirable behavior for 0!.

But what if we are asked to compute the factorial of a negative number? The method returns the same value of 1. The mathematical definition of factorial says that the function is undefined for negative values of n. So we shouldn't even compute an answer when n is negative. This is a precondition of the method that we can describe in documentation:

```
// pre : n >= 0
// post: returns n factorial (n!)
```

Adding the comment about this restriction is helpful, but what if someone calls our factorial method with a negative value anyway? The best solution is to throw an exception. The general syntax of the exception statement is as follows:

```
throw <exception>;
```

In Java, exceptions are objects. Before we can throw an exception, we have to construct an exception object using new. We normally construct the object as we are throwing it because the exception object includes information about what was going on when this error occurred. Java has a class

called IllegalArgumentException that is meant to cover a case like this where someone has passed an inappropriate value as an argument. So we can construct the exception object and include it in a throw statement as follows:

```
throw new IllegalArgumentException();
```

Of course, we only want to do this in the case where the precondition fails, so we need to include this inside of an if statement:

```
if (n < 0) {
    throw new IllegalArgumentException();
}
```

You can also include some text when you construct the exception that will be displayed when the exception is thrown:

```
if (n < 0) {
    throw new IllegalArgumentException("negative value in factorial");
}
```

Incorporating the pre/post comments and the exception code into our definition of the method, we obtain the following:

```
// pre : n >= 0
// post: returns n factorial (n!)
public static int factorial(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("negative value in factorial");
    }
    int product = 1;
    for (int i = 2; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

We don't need an else after the if that throws the exception because when an exception is thrown, it halts the execution of the method. So if someone calls the factorial method with a negative value of n, Java will never execute the code that comes after the throw statement.

We can test this with the following main method:

```
public static void main(String[] args) {
    System.out.println(factorial(-1));
}
```

When you execute this program, it stops executing with the following message:

```
Exception in thread "main" java.lang.IllegalArgumentException: negative
value in factorial
      at Factorial2.factorial(Factorial2.java:8)
      at Factorial2.main(Factorial2.java:3)
```

The message indicates that the program Factorial2 stopped running because an IllegalArgumentException was thrown. The system then shows you a backward trace of how it got there. It was in line 8 of the factorial method of the Factorial2 class. It got there because of a call in line 3 of method main of the Factorial2 class. This kind of information is very helpful in figuring out where the bugs are in your programs.

Throwing exceptions is an example of "defensive programming." We don't intend to have bugs in the programs we write, but we're only human, so we want to build in mechanisms that will give us feedback when we make mistakes. Testing the values passed to methods and throwing IllegalArgumentException when a value is not appropriate is a great way to provide that feedback.

Revisiting Return Values

In chapter 3 we saw examples of simple calculating methods that would return a value, as in this method for converting feet to miles:

```
public static double miles(double feet) {
    return feet / 5280.0;
}
```

Now that we know how to write if/else statements, we can have much more interesting examples involving return values. For example, we saw that the Math class has a method called max that returns the larger of two values. There are actually two different versions of the method, one that finds the max of two ints and one that finds the max of two doubles. Recall that this is called overloading.

Let's write our own version of the max method that returns the larger of two ints. Its header will look like this:

```
public static int max(int x, int y) {
    ...
}
```

We either want to return x or return y depending upon which one is larger. This is a perfect place to use an if/else:

```
public static int max(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

This begins by testing whether x is greater than y. If it is, it executes the first branch by returning x. If not, it executes the else branch by returning y. There are three different cases to consider: x might be larger, y might be larger, or they might be equal. The code above executes the else branch when the values are equal, but it doesn't matter which return statement is executed when they are equal.

Remember that when Java executes a return statement, the method stops executing. It's like a command to Java to "get out of this method right now." That means that this method can also be written as follows:

```
public static int max(int x, int y) {
    if (x > y) {
        return x;
    }
    return y;
}
```

This version is equivalent in behavior because the statement "return x" inside the if statement will cause Java to exit the method immediately and not to execute the "return y" statement that comes after the if. If, on the other hand, we don't enter the if, then we proceed to the statement that follows it (return y).

Whether you use the first form or the second depends somewhat on personal taste. The if/else makes it more clear that the method is choosing between two alternatives. But some people prefer the second because it is shorter. Unfortunately, we can't rely on Sun to break the tie because the Java source code expresses this using something known as the "ternary operator" (a minor detail of Java that you can read about by googling "Java ternary operator").

As another example, consider the indexOf method in the String class. If we define a variable s that stores a String:

```
String s = "four score and seven years ago";
```

We can write expressions like the following to determine where a particular character appears in the String:

```
int r = s.indexOf('r');
int v = s.indexOf('v');
```

This code sets r to 3 because that is the index of the first occurrence of the letter 'r' in the String. It sets v to 17 because that is the index of the first occurrence of the letter 'v' in the String.

The indexOf method is part of the String class, but let's see how we could write a method that performs the same task. Obviously our method would be called differently. We would have to pass it both the String and the letter, as in:

```
int r = indexOf('r', s);
int v = indexOf('v', s);
```

So the header for our method would look like this:

```
public static int indexOf(char ch, String s) {
    ...
}
```

Remember that when a method returns a value, you have to include the return type after the words "public static". In this case, we have indicated that the method returns an int because the index will be an integer. This problem can be solved rather nicely with a for loop that goes through each possible index from first to last. We can describe this in pseudocode as follows:

```
for (int i = 0; i < s.length(); i++) {
    if char is at position i, then we've found it.
}
```

To flesh this out, we have to think about how to test whether the character at position i is the one we are looking for. Remember that String objects have a method called charAt that allows us to pull out an individual character from the String. So we can refine our pseudocode as follows:

```
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == ch) {
        then we've found it.
    }
}
```

To complete this, we have to refine what to do when "we've found it". If we find the character, then we have our answer. The answer will be the current value of the variable i. And if that is the answer we want to return, then we can put a return statement there:

```
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == ch) {
        return i;
    }
}
```

To understand this code, you have to understand how the return statement works. For example, if the String s is the one from our example ("four score...") and we are searching for the character 'r', we know that when i is equal to 3 we will find that s.charAt(3) is equal to the character 'r'. That causes us to execute the return statement effectively saying:

```
return 3;
```

When a return statement is executed, we immediately exit the method. That means we would break out of the loop and return 3 as our answer. Even though the loop would normally increment i to 4 and keep going, we don't do that because we hit the return statement.

There is only one thing missing from our code. If we try to compile it as is, we get this error message from the Java compiler:

```
missing return statement
```

This error message occurs because we haven't told Java what to do if we never find the character we are searching for. In that case, we will execute the for loop in its entirety and reach the end of the method without having returned a value. That is not acceptable. If we say that the method returns an int, then we have to guarantee that every path through the method will return an int.

If we don't find the character, we want to return some kind of special value to indicate that the character was not found. You might imagine using the value 0, but 0 is a legal index for a String (the index of the first character). So the convention in Java is to return -1 if the character is not found. This is easy to add after the for loop:

```
public static int indexOf(char ch, String s) {
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == ch) {
            return i;
        }
    }
    return -1;
}
```

It may seem strange that we don't have a test for the final return statement that returns -1, but remember that the for loop tries every possible index of the String searching for the character. If the character appears anywhere in the String, then we will execute the return statement inside the loop and never get to the return statement after the loop. The only way to get to the return statement after the loop is to find that the character appears nowhere in the given String.

Common Programming Error: String Index Out of Bounds

It's very easy to forget that the last index of a String of length *n* is actually *n* - 1. Forgetting this fact can cause you to write incorrect text processing loops like this one:

```
// This version of the code has a mistake!
// The test should be  i < s.length()
public static int indexOf(char ch, String s) {
    for (int i = 0; i <= s.length(); i++) {
        if (s.charAt(i) == ch) {
            return i;
        }
    }
    return -1;
}
```

The program will crash if the loop runs past the end of the String. On the last pass through the loop, the value of variable `i` will be equal to `s.length()`, meaning that when it executes the if statement test, the code will crash. The error message resembles the following:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
    String index out of range: 11
        at java.lang.String.charAt(Unknown Source)
        at OutOfBoundsExample.indexOf(OutOfBoundsExample.java:9)
        at OutOfBoundsExample.main(OutOfBoundsExample.java:4)
```

An interesting thing about the bug in this example is that it only crashes if the String does not contain the character `ch`. If ch is contained in the String, the if test will be true for one of the legal indexes in s, so the code will return that index. Only if all characters from s have been examined without finding ch will the loop attempt its last fatal pass.

4.6 Case Study: Body Mass Index (BMI)

In recent years it has become common to compute the Body Mass Index or BMI for an individual. The Centers for Disease Control has put together a website about Body Mass Index at http://www.cdc.gov/nccdphp/dnpa/bmi/. As that site explains:

> Body Mass Index (BMI) is a number calculated from a person's weight and height. BMI provides a reliable indicator of body fatness for most people and is used to screen for weight categories that may lead to health problems.

It has also become popular to compare the statistics for two or more individuals to have a "fitness challenge" or if the numbers are for the same person, to get a sense of how BMI will vary if a person loses weight. In this section we will write a program that prompts the user for the height and weight of two individuals and that reports the overall results for the two people. Below is a sample execution for the program we want to write:

```
This program reads in data for two people
and computes their body mass index (BMI)
and weight status.

Enter next person's information:
height (in inches)? 73.5
weight (in pounds)? 230

Enter next person's information:
height (in inches)? 71
weight (in pounds)? 220.5

Person #1 body mass index = 29.930121708547368
overweight
Person #2 body mass index = 30.75014878000397
obese
```

In Chapter 1 we introduced the idea of *iterative enhancement* in which you develop a complex program in stages. Every professional programmer uses this technique, so it is important to learn to apply the idea yourself in the programs you write.

For this particular program, we eventually want to have a program that explains the program to the user and that computes results for two different people. We will also want the program to be well-structured. But we don't have to do everything all at once. In fact, if we try to do everything all at once, we are likely to be overwhelmed by the details. In writing this program, we will go through three different stages:

1. a program that computes results for just one person without an introduction and not worrying about program structure
2. a complete program that computes results for two people including an introduction but not worrying about program structure
3. a well-structured and complete program

One-person Unstructured Solution

Even the first version of the program prompts for user input, so we will need to construct a Scanner object to read from the console:

```
Scanner console = new Scanner(System.in);
```

To compute the BMI for an individual, we will need to know the height and weight of that person. This is a fairly straightforward "prompt and read" task. The only real decision here is what type of variable to use for storing the height and weight. People often talk about height and weight with whole numbers, but the question to ask is whether or not it makes sense for people to use fractions. Does anyone describe their height using half an inch? The answer is yes. Does anyone describe their weight using half a pound? Again, the answer is yes. So it makes more sense to store the values as doubles to allow people to enter either integer values or fractions.

```
System.out.println("Enter next person's information:");
System.out.print("height (in inches)? ");
double height1 = console.nextDouble();
System.out.print("weight (in pounds)? ");
double weight1 = console.nextDouble();
```

Once we have the height and width, we can compute the bmi. The CDC website says that the formula for adults for BMI is as follows:

$$\text{weight (lb)} / [\text{height (in)}]^2 \times 703$$

This is fairly easy to translate into a Java expression:

```
double bmi1 = weight1 / (height1 * height1) * 703;
```

If you look closely at the sample execution, you will see that we want to have blank lines to separate different parts of the user interaction. The introduction ends with a blank line, then there is a blank line after the "prompt and read" portion of the interaction. So adding an empty println and putting all of these pieces together, our method main looks like this so far:

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);

    System.out.println("Enter next person's information:");
    System.out.print("height (in inches)? ");
    double height1 = console.nextDouble();
    System.out.print("weight (in pounds)? ");
    double weight1 = console.nextDouble();
    double bmi1 = weight1 / (height1 * height1) * 703;
    System.out.println();

    ...
}
```

This program prompts for values and computes the BMI. Now we need to include code to report the results. We could use a println for the BMI:

```
System.out.println("Person #1 body mass index = " + bmi1);
```

This would work, but it produces output like the following:

```
Person #1 body mass index = 29.930121708547368
```

The long sequence of digits after the decimal point is distracting and it implies a level of precision that we simply don't have. It would be more appropriate and more appealing to the user to list just a few digits after the decimal point. So this would also have been a good place to use a printf:

```
System.out.printf("Person #1 body mass index = %4.2f\n", bmi1);
```

In the sample of execution we also see a report of the person's weight status. The CDC website includes the following table:

Weight Status by BMI

BMI	Weight Status
Below 18.5	Underweight
18.5 - 24.9	Normal
25.0 - 29.9	Overweight
30.0 and Above	Obese

There are four entries in this table, so we need four different println statements for the four possibilities. We will want some kind of if or if/else statements to control the four println statements. In this case, we know that we want to print exactly one of the four possibilities. Thefore, it makes most sense to use a nested if/else that ends with an else.

But what tests do we use for the nested if/else? If you look closely at this table, you will see that there are some gaps. For example, what if your BMI is 24.95? That isn't between 18.5 and 24.9 and it isn't between 25.0 and 29.9. It seems clear that the CDC intended the table to be interpreted slightly differently. They probably mean 18.5 - 24.999999 (repeating). But that would look rather odd in a table. In fact, this is a case where a nested if/else expresses this more clearly than the table does if you understand nested if/else statements:

```
if (bmi1 < 18.5) {
    System.out.println("underweight");
} else if (bmi1 < 25) {
    System.out.println("normal");
} else if (bmi1 < 30) {
    System.out.println("overweight");
} else { // bmi1 >= 30
    System.out.println("obese");
}
```

So putting this all together, we get a complete version of the first program:

```
 1  import java.util.*;
 2
 3  public class BMI1 {
 4      public static void main(String[] args) {
 5          Scanner console = new Scanner(System.in);
 6
 7          System.out.println("Enter next person's information:");
 8          System.out.print("height (in inches)? ");
 9          double height1 = console.nextDouble();
10          System.out.print("weight (in pounds)? ");
11          double weight1 = console.nextDouble();
12          double bmi1 = weight1 / (height1 * height1) * 703;
13          System.out.println();
14
15          System.out.println("Person #1 body mass index = " + bmi1);
16          if (bmi1 < 18.5) {
17              System.out.println("underweight");
18          } else if (bmi1 < 25) {
19              System.out.println("normal");
20          } else if (bmi1 < 30) {
21              System.out.println("overweight");
22          } else { // bmi1 >= 30
23              System.out.println("obese");
24          }
25      }
26  }
```

Below is a sample execution of the program:

```
Enter next person's information:
height (in inches)? 73.5
weight (in pounds)? 230

Person #1 body mass index = 29.930121708547368
overweight
```

Two-person Unstructured Solution

Now that we have a program that computes one person's BMI and weight status, let's expand it to handle two different people. Experienced programmers would probably begin by adding structure to the program before trying to make it handle two different people. But for novice programmers, it is often helpful to consider the unstructured solution first.

In turning this into a program that handles two people we notice a lot of overlap. We can copy and paste a great deal of code and make slightly modifications. For example, instead of using variables called height1, weight1 and bmi1, we change the code to use variables height2, weight2 and bmi2.

We also have to be careful to do each step in the right order. Looking at the sample execution, you'll see that the program prompts for data for both individuals first and then reports results for both. So we can't copy the entire program and simply paste a second copy. We have to rearrange the order so that all of the prompting happens first and all of the reporting happens later.

We also decided that in moving to this second stage, we will add code for the introduction. This code should appear at the beginning of the program and should include an empty println to produce a blank line to separate the introduction from the rest of the user interaction.

Combining these elements into a complete program, we get the following:

```
1  // This program finds the body mass index (BMI) for two individuals.
2
3  import java.util.*;
4
5  public class BMI2 {
6      public static void main(String[] args) {
7          System.out.println("This program reads in data for two people");
8          System.out.println("and computes their body mass index (BMI)");
9          System.out.println("and weight status.");
10         System.out.println();
11
12         Scanner console = new Scanner(System.in);
13
14         System.out.println("Enter next person's information:");
15         System.out.print("height (in inches)? ");
16         double height1 = console.nextDouble();
17         System.out.print("weight (in pounds)? ");
18         double weight1 = console.nextDouble();
19         double bmi1 = weight1 / (height1 * height1) * 703;
20         System.out.println();
21
22         System.out.println("Enter next person's information:");
23         System.out.print("height (in inches)? ");
24         double height2 = console.nextDouble();
25         System.out.print("weight (in pounds)? ");
26         double weight2 = console.nextDouble();
27         double bmi2 = weight2 / (height2 * height2) * 703;
28         System.out.println();
29
30         System.out.println("Person #1 body mass index = " + bmi1);
31         if (bmi1 < 18.5) {
32             System.out.println("underweight");
33         } else if (bmi1 < 25) {
34             System.out.println("normal");
35         } else if (bmi1 < 30) {
36             System.out.println("overweight");
37         } else { // bmi1 >= 30
38             System.out.println("obese");
39         }
40
41         System.out.println("Person #2 body mass index = " + bmi2);
42         if (bmi2 < 18.5) {
43             System.out.println("underweight");
44         } else if (bmi2 < 25) {
45             System.out.println("normal");
46         } else if (bmi2 < 30) {
47             System.out.println("overweight");
48         } else { // bmi2 >= 30
49             System.out.println("obese");
50         }
51     }
52 }
```

This program compiles and works. When we execute it, we get exactly the interaction we were looking for. But the program lacks structure. All of the code appears in main and there is significant redundancy. That shouldn't be a surprise because we created this version by using copy and paste in the editor. Whenever you find yourself using copy and paste, you should wonder whether there isn't a better way to solve the problem. Most often there is.

Two-person Structured Solution

Let's explore how static methods can improve the structure of the program. Looking at the code, you will notice a great deal of redundancy. For example, we have two code segments that look like this:

```
System.out.println("Enter next person's information:");
System.out.print("height (in inches)? ");
double height1 = console.nextDouble();
System.out.print("weight (in pounds)? ");
System.out.println();
```

The only difference between these two code segments is that the first uses variables height1/weight1/bmi1 and the second uses variables height2/weight2/bmi2. We eliminate redundancy by moving code like this into a method that we can call twice. So as a first approximation, we can turn this into a more generic form as a method:

```
public static void getBMI(Scanner console) {
    System.out.println("Enter next person's information:");
    System.out.print("height (in inches)? ");
    double height = console.nextDouble();
    System.out.print("weight (in pounds)? ");
    double weight = console.nextDouble();
    double bmi = weight / (height * height) * 703;
    System.out.println();
}
```

We have to pass in the Scanner from main. Otherwise we have made all of the variables local to this method. From main we can call this method twice:

```
getBMI(console);
getBMI(console);
```

Unfortunately, introducing this change breaks the rest of the code. If we try to compile and run the program, we find that we get error messages in main whenever we refer to variables bmi1 and bmi2.

The problem is that the method computes a bmi value that we need later in the program. We can fix this by having the method return the bmi value that it computes:

```
public double void getBMI(Scanner console) {
    System.out.println("Enter next person's information:");
    System.out.print("height (in inches)? ");
    double height = console.nextDouble();
    System.out.print("weight (in pounds)? ");
    double weight = console.nextDouble();
    double bmi = weight / (height * height) * 703;
    System.out.println();
    return bmi;
}
```

Notice that the method header now lists the return type as double. We also have to change main. We can't just call the method twice the way we would call a void method. Each call is returning a BMI result that needs to be remembered. So for each call, we have to store the result coming back from the method in a variable:

```
double bmi1 = getBMI(console);
double bmi2 = getBMI(console);
```

Study this change carefully because this technique can be one of the most challenging for novices to master. In writing the method, we have to make sure that it returns the BMI result. In writing the call, we have to make sure that we store the result in a variable so that we can access it later.

With this modification, the program would again compile and run properly. There is another obvious redundancy in the main program in that we have the same nested if/else twice. The only difference is that in one case we use the variable bmi1 and in the other case we use the variable bmi2. This is easily generalized with a parameter:

```
public static void reportStatus(double bmi) {
    if (bmi < 18.5) {
        System.out.println("underweight");
    } else if (bmi < 25) {
        System.out.println("normal");
    } else if (bmi < 30) {
        System.out.println("overweight");
    } else { // bmi >= 30
        System.out.println("obese");
    }
}
```

With this method, we can replace the code in main with two calls:

```
System.out.println("Person #1 body mass index = " + bmi1);
reportStatus(bmi1);
System.out.println("Person #2 body mass index = " + bmi2);
reportStatus(bmi2);
```

That takes care of the redundancy in the program, but we can still use static methods to improve the program to indicate structure. It is best to keep the main method short if possible to reflect the overall structure of the program. The problem breaks down into three major phases: introduction, compute bmi, report results. We already have a method for computing the bmi, but we haven't yet introduced methods for the introduction and the reporting of results. These are fairly simple to add.

There is one other method that makes sense to add to the program. We are using a formula from the CDC website for calculating the BMI of an individual given the person's height and weight. Whenever you find yourself programming a formula, it is a good idea to introduce a method for that formula so that it is easy to spot and so that it has a name.

Applying these different ideas, we end up with the following version of the program:

```
1  // This program finds the body mass index (BMI) for two individuals.  This
2  // variation includes several methods other than method main.
3
4  import java.util.*;
5
6  public class BMI3 {
7      public static void main(String[] args) {
8          giveIntro();
9          Scanner console = new Scanner(System.in);
10         double bmi1 = getBMI(console);
11         double bmi2 = getBMI(console);
12         reportResults(bmi1, bmi2);
13     }
14
15     // introduces the program to the user
16     public static void giveIntro() {
17         System.out.println("This program reads in data for two people");
18         System.out.println("and computes their body mass index (BMI)");
19         System.out.println("and weight status.");
20         System.out.println();
21     }
22
23     // prompts for one person's statistics, returning the BMI
24     public static double getBMI(Scanner console) {
25         System.out.println("Enter next person's information:");
26         System.out.print("height (in inches)? ");
27         double height = console.nextDouble();
28         System.out.print("weight (in pounds)? ");
29         double weight = console.nextDouble();
30         double bmi = BMIFor(height, weight);
31         System.out.println();
32         return bmi;
33     }
34
35     // this method contains the body mass index formula for converting the
36     // given height (in inches) and weight (in pounds) into a BMI
37     public static double BMIFor(double height, double weight) {
38         return weight / (height * height) * 703;
39     }
40
41     // reports the overall bmi values and weight status to the user
42     public static void reportResults(double bmi1, double bmi2) {
43         System.out.println("Person #1 body mass index = " + bmi1);
44         reportStatus(bmi1);
45         System.out.println("Person #2 body mass index = " + bmi2);
46         reportStatus(bmi2);
47     }
48
49     // reports the weight status for the given bmi value
50     public static void reportStatus(double bmi) {
51         if (bmi < 18.5) {
```

```
52              System.out.println("underweight");
53          } else if (bmi < 25) {
54              System.out.println("normal");
55          } else if (bmi < 30) {
56              System.out.println("overweight");
57          } else { // bmi >= 30
58              System.out.println("obese");
59          }
60      }
61  }
```

It has the same behavior as the unstructured solution, but it has a much nicer structure. The unstructured program is in some sense simpler, but the structured solution is easier to maintain if we want to expand the program or make other modifications. These structural benefits aren't so important in short programs, but they become essential as programs become longer and more complex.

Chapter Summary

- A fencepost loop executes a "loop-and-a-half" by placing part of a loop's body once before the loop begins.
- A cumulative sum loop declares a sum variable and incrementally adds to that variable's value inside the loop.
- An if statement lets you write code that will only execute if a certain condition is met. An if/else statement lets you execute one piece of code if a condition is met, and another if not. Conditions are boolean expressions and can be written using relational operators such as <, >=, and !=.
- If/else statements can be nested to test a series of conditions and execute the appropriate block of code based on which condition is true.
- Common code that appears in every branch of an if/else statement should be factored out so that it is not replicated multiple times in the code.
- Exceptions will occur if you ask a Scanner to read tokens of input that do not match what the user types.
- The == operator that tests for equality on primitive data doesn't behave the way we would expect on objects, so we test objects for equality by calling their `equals` method instead.
- You can 'throw' (generate) exceptions in your own code. This can be useful in cases where your code reaches an unrecoverable error condition, such as an invalid argument value being passed to a method.

Self-Check Problems

Section 4.1: Loop Techniques

1. What is wrong with the following code that attempts to add all numbers from 1 to a given maximum? Describe how to fix the code so that it will behave properly.

```
public static int sumTo(int n) {
    for (int i = 1; i <= n; i++) {
        int sum = 0;
        sum += i;
    }
    return sum;
}
```

2. What is wrong with the following code that attempts to print all numbers from 1 to a given maximum, separated by commas? Describe how to fix the code so that it will behave properly.

```
for (int i = 1; i <= n; i++) {
    System.out.print(i + ", ");
}
System.out.println();
```

3. Write code to produce a cumulative product by multiplying together many numbers read from the console.

Section 4.2: if/else Statements

4. Translate each of the following English statements into logical tests that could be used in an if/else statement. Write the appropriate if statement with your logical test. Assume that three int variables x, y, and z have been declared.
   - z is odd
   - z is not greater than y's square root
   - y is positive
   - one of x and y is even and the other is odd
   - y is a multiple of z
   - z is not zero
   - y is greater in magnitude than z
   - x and z are of opposite sign
   - y is a one-digit number
   - z is nonnegative
   - x is even
   - x is closer in value to y than z is

5. Given the following variable declarations:

```
int x = 4;
int y = -3;
int z = 4;
```

What are the results of the following relational expressions?

- `x == 4`
- `x == y`
- `x == z`
- `y == z`
- `x + y > 0`
- `x - z != 0`
- `y * y <= z`
- `y / y == 1`
- `x * (y + 2) > y - (y + z) * 2`

6. Consider the following Java method, which is written incorrectly:

```
// This method should return how many of its three
// arguments are odd numbers.
public static void printNumOdd(int n1, int n2, int n3) {
    int count = 0;

    if (n1 % 2 == 1) {
        count++;
    } else if (n2 % 2 == 1) {
        count++;
    } else if (n3 % 2 == 1) {
        count++;
    }

    System.out.println(count + " of the 3 numbers are odd.");
}
```

Under what cases will the method print the correct answer, and when will it print an incorrect answer? What should be changed to fix the code? Can you think of a way to write the code correctly without any if/else statements?

7. Write Java code to read an integer from the user, then prints "even" if that number is an even number or "odd" otherwise. You may assume that the user types a valid integer.

8. The following code contains a logic error:

```
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int number = console.nextInt();

if (number % 2 == 0) {
    if (number % 3 == 0) {
        System.out.println("Divisible by 6.");
    } else {
        System.out.println("Odd.");
    }
}
```

Examine the preceding code and describe a case where the code would print something that is untrue about the number that was entered, and explain why. Then correct the code so that the intent error is fixed.

9. What is wrong with the following code that attempts to return the number of factors of a given integer n? Describe how to fix the code so that it will behave properly.

```
public static int countFactors(int n) {
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {  // factor
            return i;
        }
    }
}
```

Section 4.3: Subtleties of Conditional Execution

10. The following code is poorly structured:

```
int sum = 1000;
Scanner console = new Scanner(System.in);
System.out.print("Is your money multiplied 1 or 2 times? ");
int times = console.nextInt();

if (times == 1) {
    System.out.print("And how much are you contributing? ");
    int donation = console.nextInt();
    sum = sum + donation;
    count1++;
    total = total + donation;
}
if (times == 2) {
    System.out.print("And how much are you contributing? ");
    int donation = console.nextInt();
    sum = sum + 2 * donation;
    count2++;
    total = total + donation;
}
```

Rewrite it so that it has a better structure and avoids redundancy. To simplify things, you may assume that the user always types a 1 or 2. (How would the code need to be modified if the user might type any number? How would it be modified if the user might type anything, even something that is not a number?)

11. The following code is poorly structured:

```
Scanner console = new Scanner(System.in);
System.out.print("How much will John be spending? ");
double amount = console.nextDouble();
System.out.println();

int numBills1 = (int) (amount / 20.0);
if (numBills1 * 20.0 < amount) {
    numBills1++;
}

System.out.print("How much will Jane be spending? ");
amount = console.nextDouble();
System.out.println();

int numBills2 = (int) (amount / 20.0);
if (numBills2 * 20.0 < amount) {
    numBills2++;
}

System.out.println("John needs " + numBills1 + " bills");
System.out.println("Jane needs " + numBills2 + " bills");
```

Rewrite it so that it has a better structure and avoids redundancy. You may wish to introduce a method to help capture redundant code.

12. Describe a potential problem with the following code.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your favorite color?");
String name = console.next();
if (name == "blue") {
    System.out.println("Mine, too!");
}
```

13. What is the output of the following code?

```
Point p1 = new Point(3, -2);
Point p2 = new Point(3, -2);

if (p1 == p2) {
    System.out.println("equal");
} else {
    System.out.println("non-equal");
}
```

14. What is the output of the following code?

```
Point p1 = new Point(3, -2);
Point p2 = new Point(4, 0);

if (p1.equals(p2)) {
    System.out.println("first");
}

p2.translate(-1, -2);
if (p1.equals(p2)) {
    System.out.println("second");
}

p2 = p1;
p2.translate(2, 5);
if (p1.equals(p2)) {
    System.out.println("third");
}

p2 = new Point(5, -3);
p1.translate(1, 1);
if (p1.equals(p2)) {
    System.out.println("fourth");
}
```

15. Write a piece of code that reads a shorthand text description of a color and prints the longer equivalent. Acceptable color names are B for Blue, G for Green, and R for Red. If the user types something other than B, G, or R, print an error message. Make your program case-insensitive so that the user can type an uppercase or lowercase letter. Here are some example dialogues:

```
What color do you want? B
You have chosen Blue.

What color do you want? g
You have chosen Green.

What color do you want? Bork
Unknown color: Bork
```

16. Write a piece of code that reads a shorthand text description of a playing card and prints the longhand equivalent. The shorthand description of the card is the card's rank (2 through 9, J, Q, K, or A) followed by its suit (C, D, H, or S). You should expand the shorthand into the form "(Rank) of (Suit)". You may assume that the user types valid input. Here are two example dialogues:

```
Enter a card: 9 S
Nine of Spades

Enter a card: K C
King of Clubs
```

17. The following expression in Java should equal 6.8, but it does not. Why does this occur?

```
0.2 + 1.2 + 2.2 + 3.2
```

18. The following code was intended to print a message, but it actually produces no output. Describe a way to fix the code so that the expected message will be printed.

```
double gpa = 3.2;
if (gpa * 3 == 9.6) {
    System.out.println("You earned enough credits.");
}
```

Section 4.4: Text Processing

19. Write an if statement that tests to see whether a String begins with a capital letter.

20. What is wrong with the following code, which attempts to count the number occurrences of the letter 'e' in a String, case-insensitively?

```
int count = 0;
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i).toLowerCase() == 'e') {
        count++;
    }
}
```

21. Consider a String stored in a variable named `name` that stores a person's first and last name, such as `"Marla Singer"`. Write the expression that would produce the last name followed by the first initial, such as `"Singer, M."`.

22. Write code to examine a String and determine how many of its letters come from the second half of the alphabet, that is, have values of 'n' or later. Compare case-insensitively; values of 'N' through 'Z' also count. Assume that every letter in the String is a letter.

Section 4.5: Methods with Conditional Execution

23. Consider a method `printTriangleType` that accepts three integer arguments that represent the lengths of the sides of a triangle and prints what type of triangle it is. The three types are equilateral, isosceles, and scalene. An equilateral triangle has all 3 sides the same length, an isosceles has 2 sides the same length, and a scalene has 3 sides of different lengths.

    However, there are certain integer values (or combinations of values) that would be illegal and could not represent the sides of a valid triangle. What are these values? How would you describe the precondition(s) of the `printTriangleType` method?

24. Consider a method `getGrade` that accepts an integer representing a student's grade percentage in a course, and returns that students numerical course grade between 0.0 (failing) and 4.0 (perfect). What are the preconditions of such a method?

25. The following method attempts to return the median (middle) of three integer values, but it contains logic errors. In what cases does the method return an incorrect result? How can the code be fixed?

```
public static int medianOf3(int n1, int n2, int n3) {
    if (n1 < n2) {
        if (n2 < n3) {
            return n2;
        } else {
            return n3;
        }
    } else {
        if (n1 < n3) {
            return n1;
        } else {
            return n3;
        }
    }
}
```

26. The last chapter had an exercise asking to write a method `quadratic` that would find the roots of a quadratic equation of the form $ax^2 + bx + c = 0$. The `quadratic` method was passed a, b, and c and then applied the following quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Under what conditions would this formula fail? Modify the `quadratic` method so that it will reject invalid values of a, b, or c by throwing an exception. (If you did not complete the exercise in the previous chapter, just write the method's header and the exception-throwing code.)

Exercises

1. Write a method named `fractionSum` that accepts an integer parameter N and returns as a double the sum of the first N terms of the sequence:

$$\sum_{i=1}^{n} \frac{1}{i}$$

Or in other words, the sequence,

```
1 + (1/2) + (1/3) + (1/4) + (1/5) + ...
```

You may assume that the parameter N is non-negative.

2. Write a method named `printFractionSum` that accepts an integer parameter N and prints to the console the first N terms of the sequence:

$$\sum_{i=1}^{n} \frac{1}{2^i}$$

The call `printFractionSum(1)` should print:

```
1
```

The call `printFractionSum(5)` should print:

```
1 + (1/2) + (1/4) + (1/8) + (1/16)
```

You may assume that the parameter N is non-negative. Print an entire, completed line of output.

3. Write a method named `repl` that accepts a String and a number of repetitions as parameters and returns the String concatenated that many times. For example, the call `repl("hello", 3)` returns `"hellohellohello"`. If the number of repetitions is 0 or less, an empty string is returned.

4. The following code is an incorrect loop to read numbers until the user types a valid ZIP code (a number between 1 and 99999 inclusive). What is wrong with the code? Explain the mistake and describe a way to fix it.

```
Scanner console = new Scanner(System.in);
System.out.print("Enter a ZIP code: ");
int number = 0;
while (number < 1 || number > 99999) {
    System.out.print("Invalid ZIP code.  Try again: ");
    number = console.nextInt();
}
```

5. Write a method named `printFactors` that accepts an integer as its parameter and uses a fencepost loop to print the factors of that number, separated by the word " and ". For example, the number 24's factors should print as:

```
1 and 2 and 3 and 4 and 6 and 12 and 24
```

You may assume that the number parameter's value is greater than 0, or for extra challenge, you may throw an exception if it is 0 or negative.

6. Write a method named `printLetters` that accepts a String as its parameter and uses a fencepost loop to print the letters of the String, separated by commas. For example, if the actual parameter's value is "Rabbit", your code should print the following:

```
R, a, b, b, i, t
```

Your method should print nothing if the empty string "" is passed.

7. Write a method named `pow` that accepts a base and an exponent as parameters and returns the base raised to the given power. For example, the call `pow(3, 4)` returns 3 * 3 * 3 * 3 or 81. Assume that the base and exponent are non-negative.

8. Write code that asks the user to enter numbers, then prints the smallest and largest of all the numbers typed in by the user. You may assume the user enters a valid number greater than 0 for the number of numbers to read. Here is an example dialogue:

```
How many numbers do you want to enter? 4
Number 1: 5
Number 2: 11
Number 3: -2
Number 4: 3
Smallest = -2
Largest = 11
```

9. Write a method named `printRange` that accepts two integers as arguments and prints the sequence of numbers between the two arguments. Print an increasing sequence if the first argument is smaller than the second, and otherwise print a decreasing sequence. If the two numbers are the same, that number should be printed between [and]. Here are two sample calls to `printRange`:

   ```
   printRange(2, 7);
   printRange(19, 11);
   printRange(5, 5);
   ```

   The output produced should be the following:

   ```
   [2, 3, 4, 5, 6, 7]
   [19, 18, 17, 16, 15, 14, 13, 12, 11]
   [5]
   ```

10. Write a piece of code that calculates a student's grade average. The user will type a line of input containing the student's name, then a number of scores, followed by that many integer scores. Here are two example dialogues:

    ```
    Enter a student record: Maria 5 72 91 84 89 78
    Maria's grade is 82.8

    Enter a student record: Jordan 4 86 71 62 90
    Jordan's grade is 77.25
    ```

    For example, Maria's grade is 82.8 because her average of (72 + 91 + 84 + 89 + 78) / 5 equals 82.8.

11. Write a method named `printTriangleType` that accepts three integer arguments that represent the lengths of the sides of a triangle and prints what type of triangle it is. The three types are equilateral, isosceles, and scalene. An equilateral triangle has all 3 sides the same length, an isosceles has 2 sides the same length, and a scalene has 3 sides of different lengths. Here are some example calls to `printTriangleType`:

    ```
    printTriangleType(5, 7, 7);
    printTriangleType(6, 6, 6);
    printTriangleType(5, 7, 8);
    printTriangleType(2, 18, 2);
    ```

    The output produced should be the following:

    ```
    isosceles
    equilateral
    scalene
    isosceles
    ```

12. Write a method named `numUnique` that takes three integers as parameters and that returns the number of unique integers among the three. For example, the call `numUnique(7, 31, 7)` should return 2 because the parameters represent 2 different numbers (the values 7 and 31). By contrast, the call of `numUnique(6, 6, 6)` would return 1 because there is only 1 unique number among the three parameters (the value 6).

13. Write a method named `average` that takes two integers as parameters and that returns the average of the two integers.

14. Modify your `pow` method from a previous exercise to use type double and to work correctly for negative numbers. For example, the call `pow(-4.0, 3.0)` returns -4.0 * -4.0 * -4.0 or -64.0, and the call `pow(4.0, -2.0)` returns 1 / 16 or 0.0625.

15. Write a method named `getGrade` that accepts an integer representing a student's grade percentage in a course, and returns that students numerical course grade between 0.0 (failing) and 4.0 (perfect). Assume that scores are in the range of 0 to 100 and that grades are based on the following scale:

```
Score                Grade
-------------------------
< 60                   0.0
60-62                  0.7
63                     0.8
64                     0.9
65                     1.0
...
92                     3.7
93                     3.8
94                     3.9
>= 95                  4.0
```

    For added challenge, make your method throw an `IllegalArgumentException` if the user passes a grade lower than 0 or higher than 100.

16. Write Java code that prompts the user to enter one or more words, and prints whether that String is a palindrome. A palindrome is a string that reads the same forwards as it does backwards, such as "abba" or "racecar".

    (Hint: The previous chapter had a self-check exercise asking you to write a method called `reverse` that would return a String in reversed order. If you completed this exercise, you may find it useful here.)

    For added challenge, make the code case-insensitive, so that words like "Abba" or "Madam" would be considered palindromes.

17. Write a method named `swapPairs` that accepts a String as a parameter and returns that String with each pair of adjacent letters reversed. If the String has an odd number of letters, the last letter is unchanged. For example, the call `swapPairs("example")` would return `"xemalpe"` and the call `swapPairs("hello there")` would return `"ehll ohtree"`.

18. Write a method named `wordCount` that accepts a String as its parameter and returns the number of words in the String. A word is a sequence of one or more non-space characters (any character other than ' '). For example, the call `wordCount("hello")` returns 1, the call `wordCount("how are you?")` returns 3, the call `wordCount(" this string has wide spaces ")` returns 5, and the call `wordCount(" ")` returns 0.

Programming Projects

1. Write a program that prompts for an integer and reports the factors of the integer (i.e., the numbers that go evenly into it).

2. Write a program that prompts for the lengths of the sides of a triangle and reports the three angles.

3. Write a program that prompts for a number and displays it in Roman numerals.

4. Write a program that prompts for a date (month, day, year) and reports the day of the week for that date. It might be helpful to know that January 1st, 1601 was a Monday.

5. A new tax law has just been passed by the government: the first $3,000 of income is free of tax, the next $5,000 is taxed at 10%, the next $20,000 is taxed at 20%, and the rest is taxed at 30%. Write an interactive program that prompts for a user's income and reports the corresponding tax.

6. Write a program that displays Pascal's triangle:

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5  10  10   5   1
        1   6  15  20  15   6   1
      1   7  21  35  35  21   7   1
    1   8  28  56  70  56  28   8   1
  1   9  36  84 126 126  84  36   9   1
1  10  45 120 210 252 210 120  45  10   1
```

   Use System.out.printf to format the output into fields of width 4.

7. A useful technique for catching typing errors is to use a check-digit. For example, suppose that a school assigns a six-digit number to each student. A seventh digit can be determined from the other digits, as in:

   ```
   (1 * (1st digit) + 2 * (2nd digit) + ... + 6 * (6th digit)) % 10
   ```

   When someone types in a student number, they type all seven digits. If the number is typed incorrectly, the check-digit will fail to match in 90% of the cases. Write an interactive program that prompts for a six-digit student number and reports the check digit for that number using the scheme described above.

Marty Stepp

Chapter 5
Program Logic and Indefinite Loops

Copyright © 2006 by Stuart Reges and Marty Stepp

- 5.1 The while Loop
  - A Loop to Find the Smallest Divisor
  - Sentinel Loops
  - Random Numbers
- 5.2 The boolean Type
  - Logical Operators
  - Short-Circuited Evaluation
  - boolean Variables and Flags
  - Boolean Zen
- 5.3 User Errors
  - Scanner Lookahead
  - Handling User Errors
- 5.4 Indefinite Loop Variations
  - The do/while Loop
  - Break and "forever" Loops
- 5.5 Assertions and Program Logic
  - Reasoning About Assertions
  - A Detailed Assertions Example
  - The Java assert Statement
- 5.6 Case Study: NumberGuess
  - Initial Version without Hinting
  - Randomized Version with Hinting
  - Final Robust Version

Introduction

The chapter begins by examining a new construct called a while loop that allows you to loop an indefinite number of times. Then it discusses the type boolean in greater detail and explores what are known as assertions and their relationship to understanding the logic of programs. It ends with a discussion of some other loop constructs available in Java.

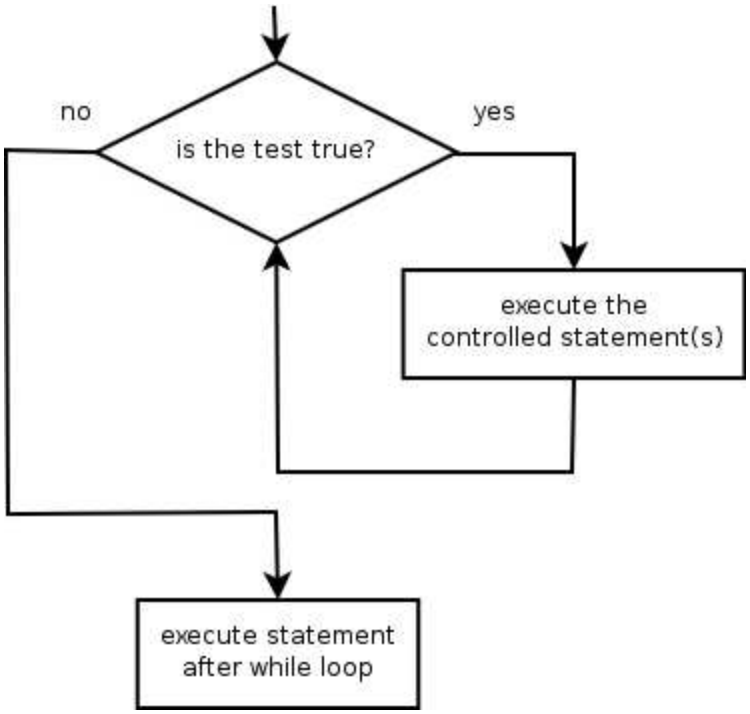
5.1 The while Loop

The for loops we have been writing since Chapter 2 are fairly simple loops that execute a predictable number of times. Recall that we call them *definite* loops because you know before the loop begins executing exactly how many times it will execute. Now we want to turn our attention to *indefinite* loops in which you don't know how many times the loops will execute. Indefinite loops come up often in interactive programs and file processing. For example, you don't know in advance how many times a user might want to play a game and you won't know before you look at a file exactly how much data it stores.

The while loop is the first indefinite loop we will study. It has the following syntax:

```
while (<test>) {
    <statement>;
    <statement>;
     ...
    <statement>;
}
```

The following diagram indicates the flow of control for the while loop. It performs its test and, if the test evaluates to true, executes the controlled statements. It repeatedly tests again and executes again if the test evaluates to true. Only when the test evaluates to false does the loop terminate.



Here is an example of a while loop.

```
int number = 1;
while (number <= 200) {
    number *= 2;
}
```

Recall that the "*=" operator says to "multiply the variable by" a certain amount, in this case multiplying the variable by 2. Thus, this loop initializes an integer variable called number to 1 and then doubles it while it is less than or equal to 200. On the surface, the if statement looks similar:

```
int number = 1;
if (number <= 200) {
    number *= 2;
}
```

The difference between the two is that the while loop executes multiple times. It loops until the test evaluates to false. The if statement executes the doubling statement once, leaving number equal to 2. The while loop executes the doubling statement until the test evaluates to false. This while loop executes the assignment statement eight times, setting number to the value 256 (the first power of 2 that is greater than 200).

The while loop tests its condition before executing the statements it controls. It performs its test at the top of the loop, as indicated in this pseudocode.

```
loop
    evaluate test.
    if test evaluates to false, then exit.
    execute controlled statements.
end of loop
```

Notice that a while loop will not execute its controlled statements if its test evaluates to false the first time the test is evaluated.

Here is a while with two statements in the loop.

```
int number = 1;
while (number <= max) {
    System.out.println("Hi there");
    number++;
}
```

This while loop is almost the same as the following for loop.

```
for (int number = 1; number <= max; number++) {
    System.out.println("Hi there");
}
```

The only difference between these two loops is the scope of the variable number. In the while loop, number is declared in the scope outside the loop. In the for loop, number is considered to be declared inside the loop.

A Loop to Find the Smallest Divisor

Suppose you want to find the smallest divisor of a number other than 1. Here are some examples of what you are looking for:

Number	Factors	Smallest Divisor
10	2 * 5	2
15	3 * 5	3
25	5 * 5	5
31	31	31
77	7 * 11	7

Here is a pseudocode description of how you might do this.

```
start divisor at 2.
while (the current value of divisor does not work) {
    increase divisor.
}
```

You don't start divisor at 1 because you are looking for the first divisor greater than 1. To modify this code you must be more explicit about what makes a divisor work. A divisor of a number has no remainder when the number is divided by it. You can rewrite this as:

```
start divisor at 2.
while (the remainder of number/divisor is not 0) {
    increase divisor.
}
```

Here is a use for the mod operator which gives the remainder for integer division. The following while loop performs this task:

```
int divisor = 2;
while (number % divisor != 0) {
    divisor++;
}
```

One problem you will undoubtedly encounter in writing your while loops is the infamous infinite loop. Consider the following code:

```
int number = 1;
while (number > 0) {
    number++;
}
```

Because number begins as a positive value and the loop makes it larger, this loop will continue indefinitely. You must be careful in formulating your while loops to avoid situations where a piece of code will never finish executing. Every time you write a while loop you should consider when and how it will finish executing.

Common Programming Error: Infinite loop

It is relatively easy to write a while loop that never terminates, causing its body to repeat infinitely. One reason it's so easy to make this mistake is that a while loop doesn't have an update step in its header like a for loop does. A correct update step is crucial because it is needed to eventually cause the loop's test to fail.

Consider the following code that tries to prompt the user for a number and repeatedly print that number divided in half until 0 is reached. This first attempt doesn't compile.

```
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");

// This code does not compile.
while (number > 0) {
    int number = console.nextInt();
    System.out.println(number / 2);
}
```

The problem with the above code is that the variable number needs to be in scope during the loop's test, so it cannot be declared inside the loop. An incorrect attempt to fix this compiler error would be to cut and paste the line initializing int number outside the loop.

```
// This code has an infinite loop.
int number = console.nextInt();   // moved out of loop

while (number > 0) {
    System.out.println(number / 2);
}
```

This second version has an infinite loop; if we ever enter the loop, we will never exit. This behavior happens because there is no update inside the while loop's body to change the value of number. If the number is greater than 0, the loop will keep printing the number and checking the loop test, which will be true every time.

The following version of the code solves the infinite loop problem. The loop contains an update step on each pass through the while loop that divides the integer in half and stores its new value. If the integer hasn't reached 0, the loop repeats.

```
// This code behaves correctly.
int number = console.nextInt();   // moved out of loop

while (number > 0) {
    number = number / 2;     // update step: divide in half and store
    System.out.println(number);
}
```

The key idea is that every while loop's body should contain code to update the terms being tested in the loop test. If the while loop test examines a variable's value, the loop body should potentially reassign a meaningful new value to that variable.

Sentinel Loops

Suppose you want to read a series of numbers from the user and compute their sum. You could ask the user in advance how many numbers to read, but that isn't always convenient. What if the user has a long list of numbers to enter? One way around this is to pick some special input value that will signal the end of input. We call this a *sentinel value*.

Sentinel

A special value that signals the end of input.

For example, you could tell the user to enter the value -1 to stop entering numbers. But how do we structure our code to make use of this sentinel? In general, we want to do the following.

```
sum = 0.
while (we haven't seen the sentinel) {
    prompt & read.
    add it to the sum.
}
```

But we don't want to add the sentinel value into our sum. This is a classic fencepost or loop-and-a-half problem. We want to prompt for and read the sentinel, but we don't want to add it to the sum.

We can use the usual fencepost solution of doing the first prompt and read before the loop and reversing the order of the two steps in the body of the loop.

```
sum = 0.
prompt & read.
while (we haven't seen the sentinel) {
    add it to the sum.
    prompt & read.
}
```

We can refine this pseudocode by introducing a variable for the number that we read from the user:

```
sum = 0.
prompt & read a value into n.
while (n is not the sentinel) {
    add n to the sum.
    prompt & read a value into n.
}
```

This translates fairly easily into Java code.

```
Scanner console = new Scanner(System.in);

int sum = 0;
System.out.print("next integer (-1 to quit)? ");
int number = console.nextInt();
while (number != -1) {
    sum += number;
    System.out.print("next integer (-1 to quit)? ");
    number = console.nextInt();
}
System.out.println("sum = " + sum);
```

When this code is executed, the interaction looks like this:

```
next integer (-1 to quit)? 34
next integer (-1 to quit)? 19
next integer (-1 to quit)? 8
next integer (-1 to quit)? 0
next integer (-1 to quit)? 17
next integer (-1 to quit)? 204
next integer (-1 to quit)? -1
sum = 282
```

Random Numbers

We often want our programs to exhibit apparently random behavior. This often comes up in game playing programs where we need our program to make up a number for the user to guess or to shuffle a deck of cards or to pick a word from a list of words for the user to guess. Programs are, by their very nature, predictable and non-random. But we can produce values that seem to be random. Such values are called pseudorandom because they are produced algorithmically.

Pseudorandom numbers

Numbers that, although they are derived from a predictable and well-defined algorithm, mimic the properties of numbers chosen at random.

Java provides several mechanisms for obtaining pseudorandom numbers. You can call the method Math.random() from the Math class to obtain a random value of type double that has the property that:

```
0.0 <= Math.random() < 1.0
```

This method provides a quick and easy way to get a random number and you can use multiplication to change the range of the numbers produced. But Java provides a class called Random that can be easier to use. It is included in the java.util package, so you'd have to include an import declaration at the beginning of your program to use it.

Random objects have several useful methods related to generating pseudorandom numbers. Each time you call one of these methods, a new random number of the requested type will be generated and returned.

Useful Methods of Random Objects

Method	Description
`nextInt()`	random integer between -2^{31} and $(2^{31} - 1)$
`nextInt(int max)`	random integer between 0 and (max - 1)
`nextDouble()`	random real number between 0.0 (inclusive) and 1.0 (exclusive)
`nextBoolean()`	random logical value of true or false

To create random numbers, you first construct a Random object:

```
Random r = new Random();
```

you can then call its method nextInt passing it a maximum integer. The number returned will be between 0 (inclusive) and the maximum (exclusive). For example, if you call nextInt(100), you will get a number between 0 and 99. You can add 1 to the number to have a range between 1 and 100.

Let's look at a simple program that picks numbers between 1 and 10 until a particular number comes up. Let's use the Random class to construct an object for generating our pseudorandom numbers:

Our loop should look something like this (where number is the value the user has asked us to generate):

```
int result;
while (result != number) {
    result = r.nextInt(10) + 1;   // random number from 1-10
    System.out.println("next number = " + result);
}
```

Notice that we have to declare the variable result outside the while loop because result appears in the while loop test. The code above has the right approach, but Java won't accept it. The code generates an error message that the variable result might not be initialized. This is an example of a loop that would need priming.

Priming a loop

Initializing variables before a loop to "prime the pump" and guarantee that we enter the loop.

We want to set the variable result to something that will cause us to enter the loop, but in some sense we don't care what value we set it to as long as it gets us into the loop. We would want to be careful not to set it to the value the user wants us to generate. We are dealing with values between 1 and 10, so we could set result to a value like -1 that is clearly outside the range of numbers we are working with. We sometimes refer to this as a "dummy" value because we don't actually process it. Later in this chapter we will see a variation of the while loop that wouldn't require this kind of priming.

The following is the complete program solution.

```
 1  import java.util.*;
 2
 3  public class Pick {
 4      public static void main(String[] args) {
 5          System.out.println("This program picks random numbers from 1 to 10");
 6          System.out.println("until a particular number comes up.");
 7          System.out.println();
 8
 9          Scanner console = new Scanner(System.in);
10          Random r = new Random();
11
12          System.out.print("Pick a number between 1 and 10--> ");
13          int number = console.nextInt();
14
15          int result = -1;
16          int count = 0;
17          while (result != number) {
18              result = r.nextInt(10) + 1;
19              System.out.println("next number = " + result);
20              count++;
21          }
22          System.out.println("Your number came up after " + count + " times");
23      }
24  }
```

Depending upon the sequence of numbers returned by the Random object, it might end up picking the given number quickly, as in this sample execution:

```
This program picks random numbers from 1 to 10
until a particular number comes up.

Pick a number between 1 and 10--> 2
next number = 7
next number = 8
next number = 2
Your number came up after 3 times
```

or it might take a while to pick the number, as in this sample execution:

```
This program picks random numbers from 1 to 10
until a particular number comes up.

Pick a number between 1 and 10--> 10
next number = 9
next number = 7
next number = 7
next number = 5
next number = 8
next number = 8
next number = 1
next number = 5
next number = 1
next number = 9
next number = 7
next number = 10
Your number came up after 12 times
```

Common Programming Error: Misusing Random object

A Random object creates a new random integer every time the nextInt method is called on it. When trying to produce a constrained random value, such as one that is odd, some students mistakenly write code such as the following:

```
// This code contains a bug.
Random r = new Random();

if (r.nextInt() % 2 == 0) {
    System.out.println("Even number: " + r.nextInt());
} else {
    System.out.println("Odd number: " + r.nextInt());
}
```

The preceding code fails in many cases because the Random object produces one random integer for use in the loop test, then another for use in whichever println statement is chosen to execute. For example, the if test might retrieve a random value of 47 from the Random object; it would see that 47 % 2 does not equal 0, so the code would proceed to the 'else' statement. The println statement would execute another call on nextInt, which would return a completely different number, say 128. The output of the code would then be the following bizarre statement:

```
Odd number: 128
```

The solution to this problem is to store the randomly created integer into a variable and only call nextInt again if another random integer is truly needed. The following code accomplishes this task:

```
// This code behaves correctly.
Random r = new Random();
int n = r.nextInt();     // save random number into a variable
if (n % 2 == 0) {
    System.out.println("Even number: " + n);
} else {
    System.out.println("Odd number: " + n);
}
```

5.2 The boolean Type

George Boole was such a good logician that Java has a data type named after him. You use the Java type boolean to describe logical true/false relationships. Recall that boolean is one of the primitive types like int, double and char.

Without realizing it, you have already used booleans. All of the control structures we have looked at--if/else statements, for loops and while loops--are controlled by expressions that specify tests. The following expression:

```
number % 2 == 0
```

is a test for divisibility by 2. It is also a boolean expression. Boolean expressions are meant to capture the concepts of truth and falsity, so it is not surprising that the domain of type boolean has only two values--true and false. The words true and false are reserved words in Java. They are the literal values of type boolean. All boolean expressions, when evaluated, will return one or the other of these literals.

To understand this better, remember what these terms mean for type int. The literals of type int include 0, 1, 2, and so on. Because these are literals of type int, you can do things like the following:

```
int number1 = 1;
int number2 = 0;
```

Consider what you can do with variables of type boolean. Suppose you define variables called test1 and test2 of type boolean. These variables can only take on two possible values--true and false. You can say:

```
boolean test1 = true;
boolean test2 = false;
```

You can also write a statement that copies the value of one boolean variable to another, as with variables of any other type.

```
test1 = test2;
```

You also know that the assignment statement can use expressions:

```
number1 = 2 + 2;
```

and that the simple tests you have been using are boolean expressions. That means you can say things like the following.

```
test1 = (2 + 2 == 4);
test2 = (3 * 100 < 250);
```

These assignment statements say, "set this boolean variable according to the truth value returned by the following test." The first statement sets the variable test1 to true, because the test evaluates to true. The second sets the variable test2 to false, because the second test evaluates to false. The parentheses are not needed, but make the statements more readable.

Many beginners don't understand these assignment statements and write code like the following.

```
if (x < y) {
    less = true;
} else {
    less = false;
}
```

This is a redundant statement. First, it evaluates the truth value of the test (x < y). If the test evaluates to true, it executes the if part and assigns less the value true. If the truth evaluates to false, is executes the else part and assigns less the value false. Since you are assigning less the truth value of the if/else test, you should do it directly.

```
less = (x < y);
```

Obviously, then, the assignment statement is one of the operations you can perform on variables of type boolean.

Logical Operators

You form complicated boolean expressions using what are known as the logical operators.

Logical Operators

Operator	Meaning	Example	Value
`&&`	and (conjunction)	`(2 == 2) && (3 < 4)`	true
`||`	or (disjunction)	`(1 < 2) || (2 == 3)`	true
`!`	not (negation)	`!(2 == 2)`	false

The not operator ("!") reverses the truth value of its operand. If an expression evaluates to true, its negation evaluates to false and vice versa. You can express this using a truth table. The following truth table has two columns, one for a variable and one for its negation. The table shows for each value of the variable, the corresponding value of the negation.

Truth Table
for Not ("!")

p	!p
true	false
false	true

In addition to the negation operator, there are two logical connectives you will use, and ("&&") and or ("||"). You use these connectives to tie two boolean expressions together, creating a new boolean expression. The truth table below shows that the and operator evaluates to true only when both of its individual operands are true:

Truth Table for And
("&&")

p	q	p && q
true	true	true
true	false	false
false	true	false
false	false	false

The truth table below shows that the or operator evaluates to true except when both operands are false.

Truth Table for Or ("||")

p	q	p || q
true	true	true
true	false	true
false	true	true
false	false	false

The Java or operator has a slightly different meaning than the English "or." In English you say, "I'll study tonight or I'll go to a movie." One or the other will be true, but not both. The or operator behaves differently. If both operands are true, the overall proposition is true.

You generally use the logical operators when what you have to say does not reduce to one test. For example, suppose you want to do something if a number is between 1 and 10. You might say:

```
if (number >= 1) {
    if (number <= 10) {
        doSomething();
    }
}
```

You can say this more easily using logical and:

```
if (number >= 1 && number <= 10) {
    doSomething();
}
```

People use the words "and" and "or" all the time. Java only allows you to use them in the strict logical sense, however. So be careful not to write code like the following:

```
// This does not compile.
if (x == 1 || 2 || 3) {
    doSomething();
}
```

In English we would read this as "x equals 1 or 2 or 3", which makes sense to us, but not to Java. Or you might say:

```
// This does not compile.
if (1 <= x <= 10) {
    doSomethingElse();
}
```

In Mathematics this expression would make sense and would test whether x is between 1 and 10 inclusive. The expression doesn't make sense in Java.

You can only use the logical and and logical or operators to combine a series of boolean expressions. Otherwise, the computer will not understand what you mean. To express the "1 or 2 or 3" idea, you would combine three different boolean expressions with logical or:

```
if (x == 1 || x == 2 || x == 3) {
    doSomething();
}
```

To express the "between 1 and 10 inclusive" idea, you would combine two boolean expressions with a logical and:

```
if (1 <= x && x <= 10) {
    doSomethingElse();
}
```

Now that you have seen the operators and, or, and not, you must again consider the precedence of operators. Section 2.3.3 has a table showing Java's precedence rules. The not operator appears at the top with the highest level of precedence. The other two logical operators have a fairly low precedence, lower than the arithmetic and relational operators but higher than the assignment operators. Between the two, the and operator has a higher level of precedence than the or operator.

These levels of precedence answer questions that arise when evaluating expressions like the following.

```
if (test1 || !test2 && test3) {
    doSomething();
}
```

Here, the computer evaluates the not first, the and second, and then the or.

Below is the precedence table once more including these new operators.

Java Operator Precedence

Description	Operators
unary operators	!, ++, --, +, -
multiplicative operators	*, /, %
additive operators	+, -
relational operators	<, >, <=, >=
equality operators	==, !=
logical and	&&
logical or	||
assignment operators	=, +=, -=, *=, /=, %=, &&=, ||=

Short-Circuited Evaluation

In this section we will explore the use of the logical operators to solve a complex programming task and we will learn about an important property of the logical operators. We are going to write a method called firstWord that takes a string as a parameter and that returns the first word in the string. To keep things simple we will adopt the convention that a String is broken up into individual words by spaces. If the String has no words at all, then the method should return an empty string. Below are a few example calls:

```
Method Call                                          Value Returned
-------------------------------------------------------------------
firstWord("four score and seven years")        "four"
firstWord("all-one-word-here")                 "all-one-word-here"
firstWord("   lots    of   space   here")      "lots"
firstWord("       ")                           ""
```

This task sounds easier than it is. We're looking to return a piece of the overall string. Remember that we can call the substring method to pull out part of a string. We pass two parameters to the substring method: the starting index of the substring and the index one beyond the end of the substring. If the overall string is stored in a variable called s, then our task basically reduces to the following steps:

```
set start to the first index of the word
set stop to the index just beyond the word
return s.substring(start, stop)
```

As a first approximation, let's assume that the starting index is 0. That won't work for strings that begin with spaces, but it will allow us to focus on the second step in the pseudocode. So consider a string that begins with "four score". If we examine the individual characters of the string and their indexes, we find the following pattern:

```
 0    1    2    3    4    5    6    7    8    9   ...
 |    |    |    |    |    |    |    |    |    |
'f'  'o'  'u'  'r'  ' '  's'  'c'  'o'  'r'  'e'  ...
```

We are assuming the first word begins at index 0. We want to set a variable stop to be the index just beyond the end of the first word. In this example, the word we want is "four" and it extends from indexes 0 through 3. So if we want the variable stop to be one beyond the end of the string, then we want to set it to index 4.

Why index 4? Because that is the first character of the string that isn't part of the word. What is special about it? It's a space. In particular, it is the first space in the string. So how do we find the first space in the string? We use a while loop. We simply start at the front of the string and loop until we get to a space:

```
set stop to 0.
while (the character at index stop is not a space) {
    stop++;
}
```

This is easily converted into Java code. Combining it with our assumption that start will be 0, we get:

```
public static String firstWord(String s) {
    int start = 0;
    int stop = 1;
    while (s.charAt(stop) != ' ') {
        stop++;
    }
    return s.substring(start, stop);
}
```

This version of the method works for many cases. It works for the String that begins with "four score", returning the string "four". But it doesn't work for all strings. It has two major limitations. We began by assuming that the string did not begin with spaces, so we know we have to fix that limitation. But it has another important limitation. It doesn't work on one-word strings. For example, if we execute it with a string like "four", it generates a StringIndexOutOfBoundsException indicating that 4 is not a legal index.

The exception occurs because our code assumes that we will eventually find a space. There is no space in the string "four". So we increment stop until it becomes equal to 4 and that causes the exception because there is no character at index 4. This is sometimes referred to as *running off the end of the string*.

To address this problem, we need to incorporate a test that that involves the length of the string. Many novices try to solve this with some combination of while and if. For example, we might write:

```
int stop = 0;
while (stop < s.length()) {
    if (s.charAt(stop) != ' ') {
        stop++;
    }
}
```

This code works for one-word strings like "four" because as soon as stop becomes equal to the length of the string, we break out of the loop. But the code doesn't work for the original multi-word cases like "four score". We end up in an infinite loop because once stop becomes equal to 4, we stop incrementing it, but we get trapped inside the loop because the test says to continue as long as stop is less than the length of the string.

The point to recognize is that this is a case where there are two different conditions that we need to use in controlling the loop. We want to continue incrementing stop only if we know that we haven't seen a space *and* that we haven't reached the end of the string. We can express that idea using the logical and operator:

```
int stop = 0;
while (s.charAt(stop) != ' ' && stop < s.length()) {
    stop++;
}
```

Unfortunately, even this test does not work. It expresses the two conditions properly because we want to make sure that we haven't reached a space and we want to make sure that we haven't reached the end of the string. But think about what happens just as we reach the end of a string. Suppose that s is "four" and stop is equal to 3. We see that the character at index 3 is not a space and we see that stop is less than the length of the string. So we increment one more time and stop becomes 4. As we come around the loop, we test whether s.charAt(4) is a space. This test throws an exception. We also test whether stop is less than 4, which it isn't, but that test comes too late to avoid the exception.

Java offers a solution for this situation. The logical operators && and || use *short-circuited evaluation*.

Short-Circuited Evaluation

The property of the logical operators `&&` and `||` that prevents the second operand from being evaluated if the overall result is obvious from the value of the first operand.

In our case, we have two different tests that we are performing and we are asking for the logical and of the two tests. If either test fails, then the overall result is false. So if the first test fails, the second test doesn't have to be performed. Because of short-circuited evaluation, we don't perform the second test at all because the overall result is obvious from the first test. Thus, the evaluation of the second test is prevented (short-circuited) by the fact that the first test fails.

So we need to reverse the order of our two tests:

```
int stop = 0;
while (stop < s.length() && s.charAt(stop) != ' ') {
    stop++;
}
```

If we run through the same scenario again with stop equal to 3, we pass both of these tests and increment stop to 4. Then as we come around the loop again, we first test to see if stop is less than s.length(). It is not, which means the test evaluates to false. As a result, Java knows that the overall expression will evaluate to false and it never evaluates the second test. This prevents the exception from occurring because we never test whether s.charAt(4) is a space.

This solution gives us a second version of the method:

```
public static String firstWord(String s)
    int start = 0;
    int stop = 0;
    while (stop < s.length() && s.charAt(stop) != ' ') {
        stop++;
    }
    return s.substring(start, stop);
}
```

But remember that we assumed that the first word starts at position 0. That won't necessarily be the case. For example, if we pass a string that begins with several spaces at the front, then it returns an empty string. We need to modify the code so that it skips any leading spaces. Accomplishing that goal requires another loop. As a first approximation, we can say:

```
int start = 0;
while (s.charAt(start) == ' ') {
    start++;
}
```

This code works for most strings, but it fails in two important cases. The loop test assumes we will find a nonspace character. What if the string is composed entirely of spaces? Then we'll never find a nonspace character. Instead, we'll simply run off the end of the string, generating a StringIndexOutOfBoundsException. And what if the string is empty to begin with? Then we get an error immediately when we ask about s.charAt(0) because there is no character at index 0.

We could decide that these cases constitute an error. After all, how can you return the first word if there is no word? So we could document a precondition that the string contains at least one nonspace character and we could even throw an exception if we find it doesn't. Another approach is to return an empty string in these cases.

We need to modify our loop to incorporate a test on the length of the string. If we add it at the end of our while loop test, we get:

```
int start = 0;
while (s.charAt(start) == ' ' && start < s.length()) {
    start++;
}
```

This code has the same flaw we saw before. It is supposed to prevent problems when start becomes equal to the length of the string, but because of the order of the tests, it will throw a StringIndexOutOfBoundsException before we reach the test on the length of the string. So these tests also have to be reversed to take advantage of short-circuited evaluation:

```
int start = 0;
while (start < s.length() && s.charAt(start) == ' ') {
    start++;
}
```

To combine this with our other code we have to change the initialization of stop. We no longer want to search from the front of the string. Instead, we need to initialize stop to be equal to start. Putting these pieces together, we get the following version of the method:

```
public static String firstWord(String s) {
    int start = 0;
    while (start < s.length() && s.charAt(start) == ' ') {
        start++;
    }
    int stop = start;
    while (stop < s.length() && s.charAt(stop) != ' ') {
        stop++;
    }
    return s.substring(start, stop);
}
```

This version works in all cases, skipping any leading spaces and returning an empty string if there is no word to return.

boolean Variables and Flags

if/else statements are controlled by a boolean test. The test can be a boolean variable or a boolean expression. Consider, for example, the following code.

```
if (number > 0) {
    System.out.println("positive");
} else {
    System.out.println("negative");
}
```

It could be rewritten as follows.

```
boolean positive = (number > 0);
if (positive) {
    System.out.println("positive");
} else {
    System.out.println("negative");
}
```

Boolean variables add to the readability of programs because they allow you to give names to tests. Consider the kind of code you would generate for a dating program. You might have some integer variables that describe certain attributes of a person: looks, to store a rough estimate of physical beauty (on a scale of 1-10); IQ, to store intelligence quotient; income, to store gross annual income; and snothers, to track intimate friends (snother is short for "significant other"). Given these variables to specify the attributes of a person, you can develop various tests of suitability. Boolean variables are useful here to give names to those tests and add greatly to the readability of the code.

```
boolean cute = (looks >= 9);
boolean smart = (IQ > 125);
boolean rich = (income > 100000);
boolean available = (snothers == 0);
boolean awesome = cute && smart && rich && available;
```

You might find occasion to use a special kind of boolean variable called a flag. Typically we use flags within loops to record error conditions or to signal completion. Different flags test different conditions. The analogy is to a referee at a game that watches for a particular illegal action and throws a flag if it happens. You sometimes hear an announcer saying, "There is a flag down on the play."

Let's introduce a flag into the cumulative sum code we saw in the previous chapter.

```
double sum = 0.0;
for (int i = 1; i <= totalNumber; i++) {
    System.out.print("    #" + i + "? ");
    double next = console.nextDouble();
    sum += next;
}
System.out.println("sum = " + sum);
```

Suppose we want to know if the sum ever goes negative. Notice that this isn't the same as whether the sum ends up being negative. The sum might switch back and forth between positive and negative. This situation is a lot like what happens with a bank account. You might make a series of deposits and withdrawals, but the bank wants to know if you overdrew your account anywhere along the way. Using a boolean flag, we can modify this loop to keep track of whether the sum ever goes negative and we can report it after the loop.

```
double sum = 0.0;
boolean negative = false;
for (int i = 1; i <= totalNumber; i++) {
    System.out.print("    #" + i + "? ");
    double next = console.nextDouble();
    sum += next;
    if (sum < 0.0) {
        negative = true;
    }
}
System.out.println("sum = " + sum);
if (negative) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}
```

Boolean Zen

In 1974 Robert Pirsig started a cultural trend with his book *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. A slew of later books copied the title so that we got "Zen and the Art of X" where X was Poker, Knitting, Writing, Foosball, Guitar, Public School Teaching, Making a Living, Falling in Love, Quilting, Stand-Up Comedy, the SAT, Flower Arrangement, Fly Tying, Systems Analysis, Fatherhood, Screenwriting, Diabetes Maintenance, Intimacy, Helping, Street Fighting, Murder, and on and on. There was even a book on *Zen and the Art of Anything*.

We join this cultural trend by discussing Zen and the Art of Type Boolean. It seems to take a while for many novices to get used to boolean expressions. Novices often write overly complex expressions involving boolean values because they don't seem to understand the simplicity that is possible when you "get" how boolean works.

For example, suppose that you want to write a method that determines whether or not a number is even. We might call the method isEven. It would take a value of type int and would return true if the int is even and false if it is not. So the method would look like this:

```
public static boolean isEven(int n) {
    ...
}
```

So how do we write the body of this method? We can use the mod operator to see if the remainder when we divide by 2 is 0. If we find that (n % 2) is 0, then we know the number is even. If not, then we know the number is not even (that it is odd). The method has a boolean return type, so we want to return the value true when (n % 2) is 0 and we want to return the value false when it is not. So we can write the method as follows:

```
public static boolean isEven(int n) {
    if (n % 2 == 0) {
        return true;
    } else {
        return false;
    }
}
```

This works, but it is more verbose than it needs to be. There is a much simpler way to write this. Think about what the code above is doing. It evaluates the expression (n % 2 == 0). That expression is of type boolean, which means that it evaluates to either true or false. The if/else says that if the expression evaluates to true, then return true, and if it evaluates to false, then return false. But why have the if/else? If we are going to return true when the expression evaluates to true and return false when it evaluates to false, then we can just return the value of the expression directly:

```
public static boolean isEven(int n) {
    return (n % 2 == 0);
}
```

Even this version can be simplified because the parentheses are not necessary, although they make it clearer exactly what is being returned. We are testing whether n % 2 equals 0 and are returning the result (true when it does equal 0, false when it does not).

Consider an analogy to integer expressions. To someone who understands boolean zen, the if/else version of this method looks as odd as the following code:

```
if (x == 1) {
    return 1;
} else if (x == 2) {
    return 2;
} else if (x == 3) {
    return 3;
} else if (x == 4) {
    return 4;
} else if (x == 5) {
    return 5;
}
```

If you always want to return the value of x, then why not just say:

```
return x;
```

A similar confusion can occur when students use boolean variables. In the last section we looked at a variation of the cumulative sum code that used a boolean variable called "negative" to keep track of whether or not the sum ever goes negative. We then used an if/else to print a message reporting the result:

```
if (negative) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}
```

Some novices would write this code as follows:

```
if (negative == true) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}
```

The comparison is unnecessary because the if/else is expecting an expression of type boolean to appear inside the parentheses. A boolean variable is already of the appropriate type, so you don't need to test whether it equals true, it either *is* true or it isn't (in which case it is false). To someone who understand boolean zen, the test above seems as redundant as saying:

```
if ((negative == true) == true) {
    ...
```

Novices also often write tests like the following:

```
if (negative == false) {
    ...
```

This makes more sense because the test is doing something useful in that it switches the meaning of the boolean variable (evaluating to true if the variable is false and evaluating to false if the variable is true). But the negation operator is designed to do this kind of switching of boolean values, so this test is better written as:

```
if (!negative) {
    ...
```

You should get used to reading the exclamation mark as "not", so this test would be read as "if not negative." To those who understand boolean zen, that is a more concise way to express this than to think about testing whether negative is equal to false.

5.3 User Errors

In the last chapter we saw that is good programming practice to think about the preconditions of a method and to mention them in the comments for the method. We also saw that in some cases we can throw exceptions if preconditions are violated.

When we write interactive programs, the simplest approach is to assume that the user will provide good input. We can then document our preconditions and throw exceptions when the user input isn't what we expected. In general, though, we would rather write programs that don't make assumptions about user input. We saw, for example, that the Scanner object can throw an exception if the user types the wrong kind of data. It's better to write programs that can deal with user errors. Such programs are referred to as being robust.

Robust

A program is said to be robust if it is able to execute even when presented with illegal data.

In this section we will explore how to write robust interactive programs. Before we can write robust code, we have to understand some special functionality of the Scanner class.

Scanner Lookahead

The Scanner class has methods that allow you to perform a test before you read something. In other words, it allows you to look before you leap. For each of the "next" methods of the Scanner class, there is a corresponding "has" method that tells you whether or not you can perform the given operation.

For example, we will often want to read an int using a Scanner object. But what if the user types something other than an int? Scanner has a method hasNextInt() that tells you whether or not reading an int is possible right now. To answer the question, the Scanner object looks at the next token and sees if it can be interpreted as an integer.

We tend to think of certain sequences of characters as being one and only one kind of thing, but when we read tokens, they can be interpreted in different ways. The following program will allow us to explore this.

```
 1  import java.util.*;
 2
 3  public class ExamineInput1 {
 4      public static void main(String[] args) {
 5          System.out.println("This program examines a token and tells you");
 6          System.out.println("the ways in which it could be read.");
 7          System.out.println();
 8
 9          Scanner console = new Scanner(System.in);
10
11          System.out.print("token? ");
12          System.out.println("  hasNextInt = " + console.hasNextInt());
13          System.out.println("  hasNextDouble = " + console.hasNextDouble());
14          System.out.println("  hasNext = " + console.hasNext());
15      }
16  }
```

Here is what happens when we enter the token "348":

```
This program examines a token and tells you
the ways in which it could be read.

token? 348
  hasNextInt = true
  hasNextDouble = true
  hasNext = true
```

As you'd expect, the call on hasNextInt() returns true, which means that we could interpret this token as an integer. But the scanner would also allow us to interpret this token as a double, which is why hasNextDouble() also returns true. But notice that hasNext() also returns true. That means that we could call the next() method to read this in as a String.

Here's another execution, this time for the token "348.2":

```
This program examines a token and tells you
the ways in which it could be read.

token? 348.2
  hasNextInt = false
  hasNextDouble = true
  hasNext = true
```

Finally, consider this execution for the token "hello":

```
This program examines a token and tells you
the ways in which it could be read.

token? hello
  hasNextInt = false
  hasNextDouble = false
  hasNext = true
```

The token "hello" can't be interpreted as an int or double. It can only be interpreted as a String.

Handling User Errors

Consider the following code fragment:

```
Scanner console = new Scanner(System.in);
System.out.print("How old are you in years? ");
int age = console.nextInt();
```

What if the user types something that is not an integer? If that happens, the Scanner will throw an exception on the call to nextInt(). We saw in the last section that we can test whether or not the next token can be interpreted as an integer with the hasNextInt() method. So we can test before reading an int whether the user has typed an appropriate value.

What if the user types something other than an integer? Then we'd want to discard the input, print out some kind of error message and prompt for a second input. We'd want this in a loop so that we'd keep discarding input and generating error messages until the user gives us legal input.

Here is a first attempt at a solution in pseudocode:

```
while (user hasn't given us an integer) {
    prompt.
    discard input.
    generate an error message.
}
read the integer.
```

This reflects what we want to do in general. We want to keep prompting, discarding and generating an error message as long as the input is illegal and then we want to read the integer when the input finally becomes legal. But we don't want to discard the input or generate an error message in that final case where the user gives us legal input. In other words, the last time through the loop we want to do just the first of these 3 steps (prompting, but not discarding and not

generating an error message). This is another classic fencepost problem and we can solve it in the usual way by putting the initial prompt before the loop and changing the order of the operations within the loop.

```
prompt.
while (user hasn't given us an integer) {
    discard input.
    generate an error message.
    prompt.
}
read the integer.
```

This pseudocode is fairly easy to turn into actual Java code:

```
Scanner console = new Scanner(System.in);
System.out.print("How old are you in years? ");
while (!console.hasNextInt()) {
    console.next();  // to discard the input
    System.out.println("That is not an integer.  Please try again.");
    System.out.print("How old are you in years? ");
}
int age = console.nextInt();
```

In fact, this is such a common operation that it is worth turning this into a static method:

```
public static int getInt(Scanner console, String prompt) {
    System.out.print(prompt);
    while (!console.hasNextInt()) {
        console.next();  // to discard the input
        System.out.println("That is not an integer.  Please try again.");
        System.out.print(prompt);
    }
    return console.nextInt();
}
```

Using this method, we can rewrite our original code to the following.

```
Scanner console = new Scanner(System.in);
int age = getInt(console, "How old are you in years? ");
```

When you execute this code, the interaction looks like this:

```
How old are you in years? what?
That is not an integer.  Please try again.
How old are you in years? 18.4
That is not an integer.  Please try again.
How old are you in years? ten
That is not an integer.  Please try again.
How old are you in years? darn!
That is not an integer.  Please try again.
How old are you in years? help
That is not an integer.  Please try again.
How old are you in years? 19
```

5.4 Indefinite Loop Variations

The while loop is the standard indefinite loop, but Java provides several alternatives. Some programmers stick to the while loop and don't bother to learn the alternatives. But the alternatives are part of the language and many programmers use them, so it is a good idea to explore how they work.

The do/while Loop

As we have seen, the while loop tests at the "top" of the loop before it executes its controlled statement. Java has an alternative known as the do/while loop that tests at the "bottom" of the loop. The do/while has the following syntax:

```
do {
    <statement>;
    ...
    <statement>;
} while (<test>);
```

For example:

```
int number = 1;
do {
    number *= 2;
} while (number <= 200);
```

This loop produces the same result as the corresponding while loop, doubling the variable number until it reaches 256, which is the first power of 2 greater than 200. But unlike the while loop, the do/while loop always executes its controlled statements at least once.

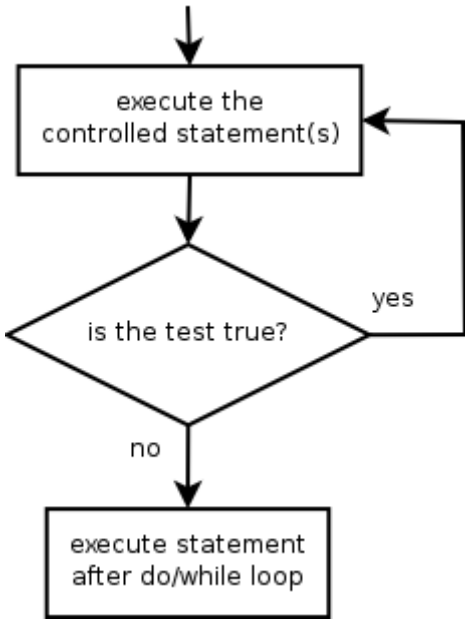
Recall that the while loop operates like this, testing before executing the controlled statement:

```
loop
    evaluate test.
    if test evaluates to false, then exit.
    execute controlled statement.
end of loop
```

The do/while loop executes like this, always executing the controlled statement at least once:

```
loop
    execute controlled statement.
    evaluate test.
    if test evaluates to false, then exit.
end of loop
```

The following diagram shows the flow of control in a do/while loop:



There are many programming problems where the do/while is the more appropriate statement. The do/while is often useful with interactive programs where you know you want to do something at least once. For example, you might have a loop that allows a user to play a game multiple times but you can be fairly sure that the user wants to play at least once. Or if you are playing a guessing game with the user, you will always have to obtain at least one guess.

We saw a situation like this earlier with the program that picked pseudorandom numbers between 1 and 10 until a certain number is picked. You know you have to pick at least once, so a do/while is appropriate. The while loop version required priming, which involved initializing a variable before the loop.

With the do/while version, we don't need to initialize because we know the loop will execute at least once:

```
 1  import java.util.*;
 2
 3  public class Pick2 {
 4      public static void main(String[] args) {
 5          System.out.println("This program picks random numbers from 1 to 10");
 6          System.out.println("until a particular number comes up.");
 7          System.out.println();
 8
 9          Scanner console = new Scanner(System.in);
10          Random r = new Random();
11
12          System.out.print("Pick a number between 1 and 10--> ");
13          int number = console.nextInt();
14
15          int result;
16          int count = 0;
17          do {
18              result = r.nextInt(10) + 1;
19              System.out.println("next number = " + result);
20              count++;
21          } while (result != number);
22          System.out.println("Your number came up after " + count + " times");
23      }
24  }
```

Notice that, as always, we use curly braces to turn multiple statements into a block. You might be tempted to move the declaration for the variable result inside the do/while loop, but that won't work because it appears in the loop test which is outside the curly braces. Also notice the semicolon that appears at the end of the do-while loop. The semicolon is a required part of the syntax for the loop.

Break and "forever" Loops

The while loop has its test at the top of the loop and the do/while has its test at the bottom of the loop. That should lead you to wonder whether it might be desirable to have the test in the middle of the loop. There is a way to accomplish this in Java, although it requires some odd looking loops.

Java has a special statement called "break" that will exit a loop. You can use it to break out of any of the loops we have seen (while, do/while, for). Loops with break statements can be difficult to understand, so using break is generally discouraged. But there is an interesting application of break to form a loop where the test occurs in the middle. The problem is that we still need to choose one of the standard loop constructs (while, do/while, for). One common choice is to form what appears to be an infinite loop with a while loop:

```
while (true) {
    <statement>;
    ...
    <statement>;
}
```

Because the boolean literal "true" always evaluates to true, this while loop appears to execute indefinitely. But you can include a test in the middle of the loop with a break statement. This technique is useful for solving the fencepost problem.

Recall that we wrote the following code to solve a fencepost problem:

```
Scanner console = new Scanner(System.in);

int sum = 0;
System.out.print("next integer (-1 to quit)? ");
int number = console.nextInt();
while (number != -1) {
    sum += number;
    System.out.print("next integer (-1 to quit)? ");
    number = console.nextInt();
}
System.out.println("sum = " + sum);
```

The code to prompt and read appears twice, once before the loop executes and once at the bottom of the loop. Using a while loop with a break, we can eliminate this redundancy:

```
Scanner console = new Scanner(System.in);

int sum = 0;
while (true) {
    System.out.print("next integer (-1 to quit)? ");
    int number = console.nextInt();
    if (number == -1) {
        break;
    }
    sum += number;
}
System.out.println("sum = " + sum);
```

Keep in mind that the while loop test is not the real test for this loop. The real test appears in the middle when we see if number is equal to -1, in which case we break. By having the test in the middle, we have a simpler solution to the "loop and a half" problem. We exit the loop on the final iteration after doing the "half" that we want.

Another form of this loop is to use a for loop that has none of the usual initialization, test and update code. The for loop still needs the parentheses and semicolons, but the rest can be empty. This leads to the following rather odd looking code:

```
Scanner console = new Scanner(System.in);

int sum = 0;
for (;;) {
    System.out.print("next integer (-1 to quit)? ");
    int number = console.nextInt();
    if (number == -1) {
        break;
    }
    sum += number;
}
System.out.println("sum = " + sum);
```

The "for (;;)" has the same effect as the "while (true)" loop. In other words, it would normally be an infinite loop. It's okay in this case because the true loop test appears in the middle of the loop with a break.

People tend to either love or hate this version of the for loop and their reason is generally the same either way. The people who hate it say, "This looks too strange." The people who love it say, "I'm glad it looks strange because it makes it clear that the real loop test is somewhere else." Some people read the "for (;;)" as "forever" and refer to these as "forever loops."

Did you Know: Controversy over break

The use of break and "forever" loops is controversial in the computer science community. A computer scientist named Edsger Dijkstra started an intense debate in 1968 when he wrote a letter to a magazine read by computer science professionals that was titled, "Goto considered harmful." He argued against a certain style of programming that was common at the time that relied heavily on "go to" commands that would cause the computer to jump from one point of program execution to another. This style of programming was called "spaghetti" code by its critics because the various goto commands would often lead to a highly unpredictable flow of control.

Dijkstra's letter included an important defense of a style of programming known as *structured programming* that has since become the dominant style of programming. All of Java, C and C++ are derived from the structured programming approach. But the break statement is a kind of "goto" that allows you to prematurely end the execution of a loop. As such, it is considered by some to be part of the "bad" programming of the past.

Many people have argued that we shouldn't be too puritanical about structured programming versus statements like break. A computer scientist named Donald Knuth, for example, has argued that for situations like a fencepost loop, the use of break makes the code simpler and, therefore, preferable.

A computer scientist named Elliot Soloway added fuel to the fire in 1983 when he published a study showing that novices can more easily learn to solve fencepost problems with forever loops and break than using while loops. His study suggests that most programmers think more easily in terms of exit conditions than in terms of continuation conditions and that forever loops with break are more intuitive to novice programmers.

There are passionate advocates on both sides of this issue and the debate has lasted for several decades, so the controversy is likely to go on indefinitely. Individual programmers will have to continue to make up their own minds about whether they like this construct or not.

5.5 Assertions and Program Logic

Logicians concern themselves with assertions.

Assertion

A declarative sentence that is either true or false.

The following are all assertions.

- 2 + 2 equals 4.
- The sun is larger than the earth.

- x > 45.
- It was raining.
- The rain in Spain falls mainly on the plain.

The following are not assertions. The first is a question and the second is a command.

- How much do you weigh?
- Take me home.

Some assertions are true or false depending upon context.

- x > 45. (This depends on x.)
- It was raining. (This depends on when and where.)

You can pin down whether they are true or false by providing a context.

- when x = 13, x > 45.
- On July 4, 1776 in Philadelphia, it was raining.

To write programs correctly and efficiently, you must learn to make assertions about your programs and to understand the contexts in which those assertions will be true. For example, if you are trying to obtain a nonnegative number from the user, you want the assertion "Number is nonnegative" to be true. What happens if you use a simple prompt and read?

```
System.out.print("Please give me a nonnegative number--> ");
double number = console.nextDouble();
// Is number nonnegative here?
```

The user can ignore your request and input a negative number anyway. In fact, users often input values that you don't expect, most often because they are confused. Given the uncertainty of user input, this particular assertion is sometimes true and sometimes false. Something later in the program may depend upon the assertion being true. For example, if you are going to take the square root of that number, you must be sure the number is nonnegative. Otherwise, you might end up with a bad result.

Using a loop we can guarantee that the number we get is nonnegative.

```
System.out.print("Please give me a nonnegative number--> ");
double number = console.nextDouble();
while (number < 0.0) {
    System.out.print("That was a negative number.  Try again--> ");
    number = console.nextDouble();
}
// Is number nonnegative here?
```

You know that number will be nonnegative after the loop; otherwise, you would not exit the while loop. As long as a user gives negative values, your program stays in the while loop and continues to prompt for input.

This doesn't mean that the number *should* be nonnegative after the loop. It means the number *will* be nonnegative. By working through the logic of the program, you can see that this is a certainty. It is an assertion of which you are sure. You could even prove it if need be. Such an assertion is called a provable assertion.

> **Provable assertion**
>
> An assertion that can be proven to be true at a particular point in program execution.

Provable assertions help to identify unnecessary bits of code. Consider these statements:

```
int x = 0;
if (x == 0) {
    System.out.println("This is what I expect.");
} else {
    System.out.println("how can that be?");
}
```

The if/else is not necessary. You know what the assignment statement does, so you know that it will set x to zero. Testing whether or not it's zero is like saying, "Before I proceed, I'm going to check that 2 + 2 equals 4." Because the if part of this if/else is always executed, you can prove that these lines of code always do the same thing.

```
int x = 0;
System.out.println("This is what I expect.");
```

This code is simpler and, therefore, better. Programs are complex enough without adding unnecessary code.

Reasoning About Assertions

The focus on assertions comes out of a field of computer science known as program verification.

> **Program Verification**
>
> A field of computer science that involves reasoning about the formal properties of programs to prove the correctness of a program.

For example, consider the properties of the simple if statement:

```
if (<test>) {
    // test is always true here
    ...
}
```

You enter the body of the if statement only if the test is true, which is why you know that it must be true at that particular point in program execution. You can draw a similar conclusion about what is true in an if/else statement:

```
if (<test>) {
    // test is always true here
    ...
} else {
    // test is never true here
    ...
}
```

We can also draw a conclusion inside the body of a while loop:

```
while (<test>) {
    // test is always true here
    ...
}
```

But in the case of the while loop, we can draw an even stronger conclusion. We know that as long as the test evaluates to true, we keep going back into the loop. That means we can conclude that after the loop is done executing, the test can no longer be true:

```
while (<test>) {
    // test is always true here
    ...
}
// test is never true here
```

The test can't be true after the loop because if it had been true, we would have executed the body of the loop again.

These observations about the properties of if statements, if/else statements and while loops provide a good start for proving certain assertions about programs. Often proving assertions requires a deeper analysis of what the code actually does. For example, suppose that you have a variable called x of type int and you execute the following if/else:

```
if (x < 0) {
    // x < 0 is always true here
    x = -x;
}
// but what about x < 0 here?
```

We wouldn't normally be able to conclude anything about x being less than 0 after the if statement, but we can if we think about the different cases. If x had been greater than or equal to 0 before the if statement, then it will still be greater than or equal to 0 after the if statement. And if x is less than 0 before the if statement, then it will be equal to -x afterwards. When x is less than 0, -x is greater than 0. Thus, in either case, we know that after the if statement executes, x will be greater than or equal to 0.

Programmers naturally apply this kind of reasoning when writing their programs. Program verification researchers are trying to figure out how to do this kind of reasoning in a formal, verifiable way.

A Detailed Assertions Example

To explore assertions further, let's work through in detail a code fragment and a set of assertions we might make about the fragment. Consider the following method:

```
public static void printCommonPrefix(int x, int y) {
    int z = 0;
    // Point A
    while (x != y) {
        // Point B
        z++;
        // Point C
        if (x > y) {
            // Point D
            x = x / 10;
        } else {
            // Point E
            y = y / 10;
        }
        // Point F
    }
    // Point G
    System.out.println("common prefix = " + x);
    System.out.println("digits discarded = " + z);
}
```

This method finds the longest sequence of leading digits that two numbers have in common. For example, the numbers 32845 and 328929343 each begin with the prefix 328. This method will compute that prefix and will also report the total number of digits that are discarded by the computation.

We will identify various assertions as being either always true, never true or sometimes true/ sometimes false at various points in program execution. The comments in the method indicate the points of interest. The assertions we will consider are:

```
x > y
x == y
z == 0
```

Normally computer scientists write assertions with mathematical notation as in "z = 0", but we will use a Java expression to distinguish this from assigning a value to the variable.

We can record our answers in a table with the words "always", "never" or "sometimes." So we want to fill in a table like the following:

```
                x > y                 x == y                z == 0
        +---------------------+---------------------+---------------------+
Point A |                     |                     |                     |
        +---------------------+---------------------+---------------------+
Point B |                     |                     |                     |
        +---------------------+---------------------+---------------------+
  ...
```

Let's start at Point A. This appears near the beginning of the method's execution:

```
public static void printCommonPrefix(int x, int y) {
    int z = 0;
    // Point A
```

The variables x and y are parameters and get their values from the call to the method. Many calls are possible, so we don't really know anything about the values of x and y. So in terms of the assertion (x > y), it could be true but it doesn't have to be. The assertion is sometimes true, sometimes false at point A. The same would be true about the assertion (x == y). It could be true depending upon what values are passed to the method, but it doesn't have to be true. In the case of the local variable z, we initialize it to 0 just before point A, so the assertion (z == 0) will always be true at that point in execution. So we would fill in the first line of the table as follows:

	x > y	x == y	z == 0
Point A	sometimes	sometimes	always

Point B appears just inside the while loop:

```
while (x != y) {
    // Point B
    z++;
    ...
}
```

We get to point B only by entering the loop, which means that the loop test has to have evaluated to true. That means that at point B it will always be true that x is not equal to y. That means that the assertion (x == y) would never be true at that point. But we don't know which of the two is larger. Therefore, the assertion (x > y) is sometimes true and sometimes false.

You might think that the assertion (z == 0) would always be true at point B because we were at point A just before being at point B, but that is not the right answer. Remember that point B is inside of a while loop. On the first iteration of the loop, we will have been at point A just before reaching point B, but not on later iterations of the loop. And if you look at the line of code just after point B, you will see that it increments z. There are no other modifications to the variable z inside the loop. Therefore, each time the body of the loop executes, z will increase by 1. So it will be 0 at point A the first time through the loop, but then it will be 1 on the second iteration, 2 on the third iteration and so forth. Therefore, the right answer for the assertion (z == 0) at point B is that it is sometimes true, sometimes false:

	x > y	x == y	z == 0
Point B	sometimes	never	sometimes

Point C is right after the increment of the variable z. There is no change to the values of x and y between point B and point C, so the same answers apply at point C for the assertions (x > y) and (x == y). The assertion that (z == 0) would never be true after the increment. z starts at 0 before the loop begins and there are no other manipulations of the variable inside the loop, so once it is incremented, it will never be 0 again. Therefore we would fill in the table for point C as follows:

```
                x > y                 x == y                z == 0
        +---------------------+---------------------+---------------------+
Point C |      sometimes      |        never        |        never        |
        +---------------------+---------------------+---------------------+
```

Points D and E are part of the if/else statement inside the loop, so we can do them as a pair. The if/else appears right after point C:

```
// Point C
if (x > y) {
    // Point D
    x = x / 10;
} else {
    // Point E
    y = y / 10;
}
```

No variables are changed between point C and points D and E. What happens is that Java performs a test and branches in one of two directions. The if/else test determines whether x is greater than y. If true, then we go to point D. If not, then we go to point E. So for the assertion (x > y), we know it is always true at point D and never true at point E. The assertion (x == y) is a little more difficult to work out. We know it can never be true at point D, but could it be true at point E? Based solely on the if/else test, the answer would be yes. But remember that at point C the assertion could never be true. The values of x and y have not changed between point C and point E, so it still can never be true.

As for the assertion (z == 0), the variable z hasn't changed between point C and points D and E and z is not included in the test. So whatever we knew about z before still holds. Therefore, the right answers to fill in for points D and E are as follows:

```
                x > y                 x == y                z == 0
        +---------------------+---------------------+---------------------+
Point D |       always        |        never        |        never        |
        +---------------------+---------------------+---------------------+
Point E |        never        |        never        |        never        |
        +---------------------+---------------------+---------------------+
```

Point F appears after the if/else. To determine the relationship between x and y at point F, we have to look at how the variables have changed. The if/else either divides x by 10 if it is the larger value or it divides y by 10 if it is the larger value. So if we think about the assertion (x > y), we have to ask whether it is possible for that to be true at point F. The answer is yes. For example, x might have been 218 and y might have been 6 before the if/else. In that case, x would now be 21 which is still larger than y. But does it have to be larger than y? Not necessarily. The values might have been reversed, in which case y is larger than x. So that assertion is sometimes true and sometimes false at point F.

What about the assertion (x == y)? We know it doesn't have to be true because we have looked at cases where x is greater than y or y is greater than x. Can it be true? We'd have to think of values of x and y that would lead to this outcome. For example, what if x had been 218 and y had been 21. Then we would have divided x by 10 and it would now be 21, which would equal y. So it also is sometimes true and sometimes false.

There was no change to z between points D/E and point F, so we simply carry our answer down from the previous columns. So we would fill in the table as follows for point F:

	x > y	x == y	z == 0
Point F	sometimes	sometimes	never

Point G appears after the while loop.

```
while (x != y) {
    ....
}
// Point G
```

We can escape the while loop only if x becomes equal to y. So at point G we know that the assertion (x == y) is always true. That means that the assertion (x > y) can never be true. The assertion (z == 0) is a little tricky. Remember that at point F it was never true. So you might imagine that at point G it can never be true. But we weren't necessarily at point F just before we reached point G. We might never have entered the while loop at all, in which case we would have been at point A just before point G. At point A the variable z was equal to 0. Therefore the right answer for this assertion is that it is sometimes true, sometimes false at point G.

	x > y	x == y	z == 0
Point G	never	always	sometimes

Putting it all together, we would fill out the table as follows:

	x > y	x == y	z == 0
Point A	sometimes	sometimes	always
Point B	sometimes	never	sometimes
Point C	sometimes	never	never
Point D	always	never	never
Point E	never	never	never
Point F	sometimes	sometimes	never
Point G	never	always	sometimes

The Java assert Statement

The concept of assertions has become so popular among software practitioners that many programming languages provide support for testing assertions. Java added support for testing assertions starting with version 1.4 of the language. The syntax for the assert statement is:

```
assert <boolean test>;
```

As in:

```
assert (x < 0);
```

Parentheses have been included here to make the boolean expression very clear, but the parentheses are not required.

Programmers often build assertions into their programs to have the computer check to make sure that all of the programmer's assumptions are correct. In general, we expect these assertions to succeed. When the assertion fails, that signals a problem. It means that the programmer has a logic error that is preventing the assumptions from holding true. If the boolean test in the assert statement evaluates to false, we say that the assertion fails. When an assertion fails, an exception is thrown that stops the program from executing.

Testing of assertions can be expensive, so Java gives you the ability to control whether assertions are enabled or disabled. That way you can enable assertion checking while you are developing and testing a program to make sure it works properly but you can disable assertion checking when you're fairly confident that the program works and you want to speed up the program. By default, assertion checking is disabled. You can enable assertion checking by giving the "-ea" option when you run your Java program.

5.6 Case Study: NumberGuess

Combining indefinite loops, the ability to check for user errors, and random numbers, it's possible to create guessing games where the computer thinks of random numbers and the user tries to guess them. Let's consider an example game with the following rules:

The computer thinks of a random two digit number but keeps it secret from the player. We'll count numbers that begin with 0, so the acceptable range of numbers here is 0 through 99 inclusive. The player will try to guess the computer's number. If the player guesses correctly, the program will report the number of guesses needed.

To make the game more interesting, the computer will give the player hints for every incorrect guess. Specifically, the computer will tell the player how many digits from the guess are contained in the correct answer. The order of the digits doesn't affect how many are matching. For example, if the correct number is 57 and the player guesses 73, the computer will report 1 matching digit because the correct answer contains a 7. If the player next guesses 75, the computer will report 2 matching digits. At this point the player knows that the computer's number must be 57, because 57 is the only two-digit number whose digits match 75's.

Since the player will be doing a lot of console input, it's likely that they might type an incorrect number or a non-numeric token by mistake. We'd like our guessing game program to be robust against user input errors.

Initial Version without Hinting

In previous chapters we've talked about the idea of iterative enhancement. Since this is a challenging program, we'll tackle it in stages. One of the hardest parts of the program is giving correct hints to the player when a guess is wrong. For now, we'll simply write a game that tells the player whether they are correct or incorrect on each guess, and once the game is done, reports the number of guesses needed. We won't be robust against user input errors yet; that can be added later. To further simplify the game, rather than having the computer choose a random number, we'll fix the number at a known value so that the code can be tested more easily.

Since it's not known exactly how many tries the player will need to guess the number, it seems that the main loop for this game will be a while loop. It's tempting to want to write the code in the following way, to match the following pseudo-code:

```
// flawed number guess pseudo-code
think of a number.
while (user has not guessed the number) {
    prompt and read a guess.
    report whether the guess was correct or incorrect.
}
```

However, the problem with the preceding pseudo-code is that you can't start the while loop if you don't have any guess value from the player yet. The following code doesn't compile, because the variable guess isn't initialized when the loop begins:

```
// This code doesn't compile.
int numGuesses = 0;
int number = 42;   // computer always picks same number
int guess;

while (guess != number) {
    System.out.print("Your guess? ");
    guess = console.nextInt();
    numGuesses++;

    System.out.println("Incorrect.");
}

System.out.println("You got it right in " + numGuesses + " tries.");
```

We could try to solve the problem with the preceding code by priming the value of the guess variable. We'd have to set it to a value other than the correct number, so that the loop would enter.

However, a fencepost "loop-and-a-half" solution will work even better. It turns out that the game's main guess loop is a fencepost loop, because after each incorrect guess we must print an "Incorrect" message (and later a hint) to the player. For n guesses, there are n-1 hints. Recall the following general pseudo-code for fencepost loops:

```
plant a post.
for (the length of the fence) {
    attach some wire.
    plant a post.
}
```

This particular problem is an indefinite fencepost using a while loop. A more specific pseudo-code is the following. The "posts" are the prompts for guesses, and the "wires" are the "Incorrect" messages.

```
// specific number guess pseudo-code
think of a number.
ask for the player's initial guess.

while (the guess is not the correct number) {
    inform the player that they were incorrect, and
    ask for another guess.
}

report the number of guesses needed.
```

The pseudo-code leads us to write the following Java program. Note that the computer always picks the value 42 for now.

```
 1  import java.util.*;
 2
 3  public class NumberGuess1 {
 4      public static void main(String[] args) {
 5          Scanner console = new Scanner(System.in);
 6          int number = 42;   // computer always picks same number
 7
 8          System.out.print("Your guess? ");
 9          int guess = console.nextInt();
10          int numGuesses = 1;
11
12          while (guess != number) {
13              System.out.println("Incorrect.");
14              System.out.print("Your guess? ");
15              guess = console.nextInt();
16              numGuesses++;
17          }
18
19          System.out.println("You got it right in " + numGuesses + " tries.");
20      }
21  }
```

A do/while loop and a while loop with priming would also do the trick at this stage, but in the next step of the program, we'll add new features that work better with the loop solution shown.

We can test our initial program to verify the code we've written so far. A sample dialogue looks like this:

```
Your guess? 65
Incorrect.
Your guess? 12
Incorrect.
Your guess? 34
Incorrect.
Your guess? 42
You got it right in 4 tries.
```

Randomized Version with Hinting

Now that we've tested to make sure our main game loops, let's first make the game random by choosing a random value between 0 and 99 inclusive. To do so, we'll create a Random object and use its nextInt, specifying the maximum value. Remember that the value passed to Random's nextInt should be one more than the desired maximum, so we'll pass 100.

```
// pick a random number between 0 and 99 inclusive
Random rand = new Random();
int number = rand.nextInt(100);
```

The next important feature our final game should have is the hint given to the player when an incorrect guess is made. The tricky part is figuring out how many digits match between the correct number and the player's guess. Since this code is non-trivial to write, let's make a helping method named matches that does this work for us. In order to figure this out, the matches method needs the guess and correct number as parameters, and it will return the number of matching digits. Therefore its header should look like this:

```
public static int matches(int number, int guess) {
    ...
}
```

Our algorithm must count the number of matching digits. Either digit from the guess can match either digit from the correct number. Since the digits are somewhat independent -- that is, whether the ones digit of the guess matches is independent of whether the tens digit matches -- we should use sequential ifs rather than if/else to represent these conditions.

The digit-matching algorithm does have a special case. If the player guesses a number with two of the same digit such as 33, and if that digit is contained in the correct answer such as 37, it would be misleading to report that 2 digits matched. A better behavior would be to report one matching digit. Because of this case, our algorithm needs to check whether the guess contains two of the same digit, and only consider the second digit of the guess a match if it is unique from the first.

Here is a pseudo-code for the algorithm:

```
matches = 0.
if (the first digit of the guess matches
      either digit of the correct number) {
    we have found one match.
}

if (the second digit of the guess is different from the first digit,
      AND it matches either digit of the correct number) {
    we have found another match.
}
```

In order to compare digits, we'll need to be able to split the correct number and the guess into their two digits. Using the division and remainder operations, we can express the digits of any two-digit number n as `n / 10` for the tens digit and `n % 10` for the ones digit.

Let's write the statement that tries to match the ones digit of the guess against a digit of the correct answer. Since the guess's digit can match either of the correct number's digits, we'll use an or test with the || operator.

```
int matches = 0;

// check the first digit for a match
if (guess / 10 == number / 10 || guess / 10 == number % 10) {
    matches++;
}
```

The statement that tries to match the tens digit of the guess against the correct answer is slightly more tricky because of the special case described above. We'll account for this by only counting the second digit as a match if it is unique, AND it matches a digit from the correct number.

```
// check the second digit for a match
if (guess / 10 != guess % 10 &&
    (guess % 10 == number / 10 || guess % 10 == number % 10)) {
    matches++;
}
```

The following second version of the program uses the hinting code we've just written. It also adds the randomly chosen number and a brief introduction to the program.

```
 1  // Two-digit number-guessing game with hinting.
 2  import java.util.*;
 3
 4  public class NumberGuess2 {
 5      public static void main(String[] args) {
 6          System.out.println("Try to guess my two-digit number, and I'll");
 7          System.out.println("tell you how many digits from your guess");
 8          System.out.println("appear in my number.");
 9          System.out.println();
10
11          Scanner console = new Scanner(System.in);
12
13          // pick a random number from 0 to 99 inclusive
14          Random rand = new Random();
15          int number = rand.nextInt(100);
16
17          // get first guess
18          System.out.print("Your guess? ");
19          int guess = console.nextInt();
20          int numGuesses = 1;
21
22          // give hints until correct guess is reached
23          while (guess != number) {
24              int numMatches = matches(number, guess);
25              System.out.println("Incorrect (hint: " + numMatches + " digits match)");
26              System.out.print("Your guess? ");
27              guess = console.nextInt();
28              numGuesses++;
29          }
30
31          System.out.println("You got it right in " + numGuesses + " tries.");
32      }
33
34      // Reports a hint about how many digits from the given guess
```

```
35      // match digits from the given correct number.
36      public static int matches(int number, int guess) {
37          int numMatches = 0;
38
39          if (guess / 10 == number / 10 ||
40              guess / 10 == number % 10) {
41              numMatches++;
42          }
43
44          if (guess / 10 != guess % 10 &&
45              (guess % 10 == number / 10 ||
46               guess % 10 == number % 10)) {
47              numMatches++;
48          }
49
50          return numMatches;
51      }
52  }
```

The following is a sample log of execution:

```
Try to guess my two-digit number, and I'll
tell you how many digits from your guess
appear in my number.

Your guess? 13
Incorrect (hint: 0 digits match)
Your guess? 26
Incorrect (hint: 0 digits match)
Your guess? 78
Incorrect (hint: 1 digits match)
Your guess? 79
Incorrect (hint: 1 digits match)
Your guess? 70
Incorrect (hint: 2 digits match)
Your guess? 07
You got it right in 7 tries.
```

Final Robust Version

The last major change we'll make to our program is to make it robust against invalid user input. There are two types of bad input we may see:

1. Non-numeric tokens
2. Numbers outside the range of 0-99

Let's deal with these cases one at a time. Recall the getInt method from earlier in the chapter. It repeatedly prompts the user for input until an integer is typed:

```
// Re-prompts until a number is entered.
public static int getInt(Scanner console, String prompt) {
    System.out.print(prompt);
    while (!console.hasNextInt()) {
        console.next();
        System.out.println("Not an integer; try again.");
        System.out.print(prompt);
    }
    return console.nextInt();
}
```

We can make use of getInt to get an integer between 0 and 99. We'll repeatedly call getInt until the integer returned is within the acceptable range.

```
// Re-prompts until a number in the proper range is entered.
// post: guess is between 0 and 99
public static int getGuess(Scanner console) {
    int guess = getInt(console, "Your guess? ");
    while (guess < 0 || guess >= 100) {
        System.out.println("Out of range; try again.");
        guess = getInt(console, "Your guess? ");
    }

    return guess;
}
```

Now, whenever we want to read user input in the main program, we'll call getGuess. It's useful to separate the input prompting in this way, to make sure that we don't accidentally count invalid inputs as guesses.

The final version of our code is the following:

```
1  // Robust two-digit number-guessing game with hinting.
2  import java.util.*;
3
4  public class NumberGuess3 {
5      public static void main(String[] args) {
6          giveIntro();
7          Scanner console = new Scanner(System.in);
8
9          // pick a random number from 0 to 99 inclusive
10         Random rand = new Random();
11         int number = rand.nextInt(100);
12
13         // get first guess
14         int guess = getGuess(console);
15         int numGuesses = 1;
16
17         // give hints until correct guess is reached
18         while (guess != number) {
19             int numMatches = matches(number, guess);
20             System.out.println("Incorrect (hint: " + matches + " digits match)");
21             guess = getGuess(console);
22             numGuesses++;
23         }
24
25         System.out.println("You got it right in " + numGuesses + " tries.");
26     }
27
28     public static void giveIntro() {
29         System.out.println("Try to guess my two-digit number, and I'll");
30         System.out.println("tell you how many digits from your guess");
31         System.out.println("appear in my number.");
32         System.out.println();
33     }
34
35     // Returns number of matching digits between the two numbers.
36     // pre: number and guess are unique two-digit numbers
37     public static int matches(int number, int guess) {
38         int numMatches = 0;
39
40         if (guess / 10 == number / 10 ||
41             guess / 10 == number % 10) {
42             numMatches++;
43         }
44
45         if (guess / 10 != guess % 10 &&
46             (guess % 10 == number / 10 ||
47              guess % 10 == number % 10)) {
48             numMatches++;
49         }
50
51         return numMatches;
52     }
53
54     // Re-prompts until a number in the proper range is entered.
55     // post: guess is between 0 and 99
56     public static int getGuess(Scanner console) {
57         int guess = getInt(console, "Your guess? ");
58         while (guess < 0 || guess >= 100) {
59             System.out.println("Out of range; try again.");
60             guess = getInt(console, "Your guess? ");
```

```
61          }
62
63          return guess;
64      }
65
66      // Re-prompts until a number is entered.
67      public static int getInt(Scanner console, String prompt) {
68          System.out.print(prompt);
69          while (!console.hasNextInt()) {
70              console.next();
71              System.out.println("Not an integer; try again.");
72              System.out.print(prompt);
73          }
74          return console.nextInt();
75      }
76  }
```

The following is a sample log of execution, demonstrating the new input robustness:

```
Try to guess my two-digit number, and I'll
tell you how many digits from your guess
appear in my number.

Your guess? 12
Incorrect (hint: 0 digits match)
Your guess? okay
Not an integer; try again.
Your guess? 34
Incorrect (hint: 1 digits match)
Your guess? 35
Incorrect (hint: 1 digits match)
Your guess? 67
Incorrect (hint: 0 digits match)
Your guess? 89
Incorrect (hint: 0 digits match)
Your guess? 03
Incorrect (hint: 2 digits match)
Your guess? 300
Out of range; try again.
Your guess? 30
You got it right in 7 tries.
```

Notice that we're careful to comment our code to document relevant preconditions and postconditions of our methods. The precondition of the matches method is that the two parameters are unique two-digit numbers. The postcondition of our new getGuesses method is that it returns a guess between 0 and 99 inclusive.

If we wanted to further assure ourselves of the correctness of our robust input code and the validity of our getGuess postcondition, we could use an assert statement in places where we read user input.

```
// program will terminate if our getGuess code returns a bad value
guess = getGuess(console);
assert (guess >= 0 && guess <= 99);
```

Chapter Summary

- Java has a while loop in addition to its for loop. The while loop can be used to write indefinite loops that keep executing until some condition fails.
- One kind of fencepost loop is called a sentinel loop, which repeatedly processes input until a special value is seen, but must not accidentally process the special value.
- A robust program checks for errors in user input. Better robustness can be achieved by looping and re-prompting the user when bad input is typed. The Scanner has methods like hasNextInt to help check for valid input.
- Priming a loop means setting the values of variables that will be used in a loop test, so that the test will be sure to succeed the first time, and the loop will execute.
- Java can generate pseudorandom numbers using objects of the Random class.
- The boolean primitive type represents logical values of either true or false. Boolean expressions are used as tests in if statements and loops. Boolean expressions can use relational operators such as < or != , as well as logical operators such as && or ! . Boolean variables (sometimes called "flags") can store boolean values and can be used as loop tests.
- The do/while loop is a variation on the while loop that performs its loop test at the end of the loop body. A do/while loop is guaranteed to execute its body at least once.
- A "forever" loop can be written to perform its loop test in the middle of its loop body. A forever loop has a test that is always true, and it can be exited by writing a special 'break' statement in the middle of the body.
- Assertions are logical statements about a particular point in a program. Assertions are useful for proving properties about how a program will execute. Two useful types of assertions are preconditions and postconditions, which are claims about what will be true before and after a method executes.

Self-Check Exercises

Section 5.1: The while Loop

1. For each of the following while loops, how many times will the loop execute its body? Remember that "zero", "infinity", and "unknown" are legal answers. Also, what is the output of the code?

```
int x = 1;
while (x < 100) {
    System.out.print(x + " ");
    x += 10;
}
```

```
int x = 2;
while (x < 200) {
    System.out.print(x + " ");
    x *= x;
}
```

```
int max = 10;
while (max < 10) {
    System.out.println("count down: " + max);
    max--;
}
```

```
String word = "a";
while (word.length() < 10) {
    word = "b" + word + "b";
}
System.out.println(word);
```

```
int x = 250;
while (x % 3 != 0) {
    System.out.println(x);
}
```

```
int x = 100;
while (x > 0) {
    System.out.println(x / 10);
    x = x / 2;
}
```

2. Convert each of the following for loops into an equivalent while loop.

   - ```
     for (int n = 1; n <= max; n++) {
         System.out.println(n);
     }
     ```

   - ```
     int total = 25;
     for (int number = 1; number <= (total / 2); number++) {
         total = total - number;
         System.out.println(total + "  " + number);
     }
     ```

   - ```
     for (int i = 1; i <= 2; i++) {
         for (int j = 1; j <= 3; j++) {
             for (int k = 1; k <= 4; k++) {
                 System.out.print("*");
             }
             System.out.print("!");
         }
         System.out.println();
     }
     ```

   - ```
     int number = 4;
     for (int count = 1; count <= number; count++) {
         System.out.println(number);
         number = number / 2;
     }
     ```

3. Consider the following method:

```
public static void mystery(int x) {
    int y = 1;
    int z = 0;
    while (2 * y <= x) {
        y *= 2;
        z++;
    }
    System.out.println(y + " " + z);
}
```

   For each call below, indicate what output is produced.

```
mystery(1);
mystery(6);
mystery(19);
mystery(39);
mystery(74);
```

4. Consider the following method:

```
public static void mystery(int x) {
    int y = 0;
    while (x % 2 == 0) {
        y++;
        x /= 2;
    }
    System.out.println(x + " " + y);
}
```

For each call below, indicate what output is produced.

```
mystery(19);
mystery(42);
mystery(48);
mystery(40);
mystery(64);
```

5. Write a sentinel loop that repeatedly prompts the user for numbers. Once the number -1 is typed, the maximum and minimum of all numbers typed are displayed. For example:

```
Type a number (or -1 to stop): 5
Type a number (or -1 to stop): 2
Type a number (or -1 to stop): 17
Type a number (or -1 to stop): 8
Type a number (or -1 to stop): -1
Maximum was 17
Minimum was 2
```

If -1 is the first number typed, you should not print any maximum or minimum. For example:

```
Type a number (or -1 to stop): -1
```

6. Write a method named `zeroDigits` that accepts an integer parameter and returns the number of digits in the number that have the value 0. For example, the call `zeroDigits(5024036)` should return 2 and `zeroDigits(743)` should return 0. The call `zeroDigits(0)` should return 1.

7. Write Java code that reads an integer from the user and prints all of the odd digits in the number in their original order, followed by all of the even digits in the number in their original order. Here is an example dialogue:

```
Type a number: 8675309
7539860
```

8. What range of values can each variable a, b, c, d, and e have?

```
Random rand = new Random();
int a = rand.nextInt(100);
int b = rand.nextInt(20) + 50;
int c = rand.nextInt(20 + 50);
int d = rand.nextInt(100) - 20;
int e = rand.nextInt(10) * 4;
```

9. Write code that generates a random integer between 0 and 10 inclusive.

10. Write code that generates a random odd integer (not divisible by 2) between 50 and 99 inclusive.

Section 5.2: The boolean Type

11. Given the following variable declarations:

```
int x = 27;
int y = -1;
int z = 32;
boolean b = false;
```

What is the value of each of the following boolean expressions?

```
!b
b || true
(x > y) && (y > z)
(x == y) || (x <= z)
!(x % 2 == 0)
(x % 2 != 0) && b
b && !b
b || !b
(x < y) == b
!(x / 2 == 13) || b || (z * 3 == 96)
(z < x) == false
!((x > 0) && (y < 0))
```

12. Write a method named `isVowel` that accepts a character as input and returns true if that character is a vowel: a, e, i, o, or u. For extra challenge, make your method case-insensitive.
13. The following code attempts to examine a number and return whether that number is prime (has no factors other than 1 and itself). A boolean flag named prime is used. However, the boolean logic is not implemented correctly, so the method does not always return the correct answer. In what cases does the method report an incorrect answer? How can the code be changed so that it will always return a correct result?

```
public static boolean isPrime(int n) {
    boolean prime = true;
    for (int i = 2; i < n; i++) {
        if (n % i == 0) {
            prime = false;
        } else {
            prime = true;
        }
    }

    return prime;
}
```

14. The following code attempts to examine a String and return whether it contains a given letter. A boolean flag named found is used. However, the boolean logic is not implemented correctly, so the method does not always return the correct answer. In what cases does the method report an incorrect answer? How can the code be changed so that it will always return a correct result?

```
public static boolean contains(String str, char ch) {
    boolean found = false;
    for (int i = 0; i < str.length(); i++) {
        if (str.charAt(i) == ch) {
            found = true;
        } else {
            found = false;
        }
    }

    return found;
}
```

15. Using "boolean zen", write an improved version of the following method, which returns whether the given String starts and ends with the same character:

```
public static boolean startEndSame(String str) {
    if (str.charAt(0) == str.charAt(str.length() - 1)) {
        return true;
    } else {
        return false
    }
}
```

16. Using "boolean zen", write an improved version of the following method, which returns whether the given number of cents would require any pennies when making change:

```
public static boolean hasPennies(int cents) {
    boolean nickelsOnly = (cents % 5 == 0);
    if (nickelsOnly == true) {
        return false;
    } else {
        return true;
    }
}
```

17. Consider the following method:

```
public static int mystery(int x, int y) {
    while (x != 0 && y != 0) {
        if (x < y) {
            y -= x;
        } else {
            x -= y;
        }
    }
    return x + y;
}
```

For each call below, indicate what value is returned.

```
mystery(3, 3)
mystery(5, 3)
mystery(2, 6)
mystery(12, 18)
mystery(30, 75)
```

Section 5.3: User Errors

18. The following code is not robust against invalid user input. Describe how to change the code so that it will not proceed until the user has entered a valid age and GPA. (Assume that any int is a legal age and that any double is a legal GPA.)

```
Scanner console = new Scanner(System.in);
System.out.print("Type your age: ");
int age = console.nextInt();

System.out.print("Type your GPA: ");
double gpa = console.nextDouble();
```

    For added challenge, modify the code so that it rejects invalid ages (for example, numbers less than 0) and GPAs (say, numbers less than 0.0 or greater than 4.0).

19. What is the output of the following code:

    - when the user types `Jane` ?
    - when the user types `56` ?
    - when the user types `56.2` ?

```
Scanner console = new Scanner(System.in);
System.out.print("Type something for me! ");

if (console.hasNextInt()) {
    int number = console.nextInt();
    System.out.println("Your IQ is " + number);
} else if (console.hasNext()) {
    String token = console.next();
    System.out.println("Your name is " + token);
}
```

20. Write a piece of code that prompts the user for a number and then prints a different message depending on whether the number was an integer or a real number. Here are two example dialogues:

```
Type a number: 42.5
You typed the real number 42.5

Type a number: 3
You typed the integer 3
```

21. Write code that prompts for three integers and then averages them and prints the average. Make your code robust against invalid input. (You may want to use the getInt method shown in this chapter.)

Section 5.4: Indefinite Loop Variations

22. For each of the following do/while loops, how many times will the loop execute its body? Remember that "zero", "infinity", and "unknown" are legal answers. Also, what is the output of the code?

```
int x = 1;
do {
    System.out.print(x + " ");
    x += 10;
} while (x < 100);
```

```
int x = 2;
do {
    System.out.print(x + " ");
    x *= x;
} while (x < 200);
```

```
int max = 10;
do {
    System.out.println("count down: " + max);
    max--;
} while (max < 10);
```

```
String word = "a";
do {
    word = "b" + word + "b";
} while (word.length() < 10);
System.out.println(word);
```

```
int x = 250;
do {
    System.out.println(x);
} while (x % 3 != 0);
```

```
int x = 100;
do {
    System.out.println(x / 10);
    x = x / 2;
} while (x > 0);
```

```
int x = 100;
do {
    System.out.println(x);
    x = x / 2;
} while (x % 2 == 0);
```

```
String str = "/\\";
do {
    str += str;
} while (str.length() < 10);
System.out.println(str);
```

23. Write a do/while loop that repeatedly prints a message until the user tells the program to stop. The do/while is appropriate because the message should always be printed at least one time, even if the user says "n" after the first message appears.

```
She sells seashells by the seashore.
Do you want to hear it again? y
She sells seashells by the seashore.
Do you want to hear it again? y
She sells seashells by the seashore.
Do you want to hear it again? n
```

24. Write a do/while loop that repeatedly prints random numbers between 0 and 1000 until a number above 900 is printed. At least one line of output is always printed, even if the first random number is above 900.

```
Random number: 235
Random number: 15
Random number: 810
Random number: 147
Random number: 915
```

25. A previous question asked you to write a sentinel loop that read numbers from the console until the user typed -1 and then printed the maximum and minimum number. Rewrite this code to use a "forever" loop with a break statement. Below is a sample execution:

```
Type a number (or -1 to stop): 5
Type a number (or -1 to stop): 2
Type a number (or -1 to stop): 17
Type a number (or -1 to stop): 8
Type a number (or -1 to stop): -1
Maximum was 17
Minimum was 2
```

Section 5.5: Assertions and Program Logic

26. Identify the various assertions below as being either always true, never true or sometimes true/sometimes false at various points in program execution. The comments in the method below indicate the points of interest.

```
public static int mystery(Scanner console, int x) {
    int y = console.nextInt();
    int count = 0;

    // Point A
    while (y < x) {
        // Point B
        if (y == 0) {
            count++;
            // Point C
        }

        y = console.nextInt();
        // Point D
    }

    // Point E
    return count;
}
```

Categorize each assertion at each point below with either ALWAYS, NEVER or SOMETIMES.

	y < x	y == 0	count > 0
Point A			
Point B			
Point C			
Point D			
Point E			

27. Identify the various assertions below as being either always true, never true or sometimes true/sometimes false at various points in program execution. The comments in the method below indicate the points of interest.

```
public static int mystery(int n) {
    Random r = new Random();
    int a = r.nextInt(3) + 1;
    int b = 2;

    // Point A
    while (n > b) {
        // Point B
        b = b + a;

        if (a > 1) {
            n--;

            // Point C
            a = r.nextInt(b) + 1;
        } else {
            a = b + 1;
            // Point D
        }
    }

    // Point E
    return n;
}
```

Categorize each assertion at each point below with either ALWAYS, NEVER or SOMETIMES.

	n > b	a > 1	b > a
Point A			
Point B			
Point C			
Point D			
Point E			

28. Identify the various assertions below as being either always true, never true or sometimes true/sometimes false at various points in program execution. The comments in the method below indicate the points of interest.

```
public static int mystery(Scanner console) {
    int prev = 0;
    int count = 0;
    int next = console.nextInt();
    // Point A
    while (next != 0) {
        // Point B
        if (next == prev) {
            // Point C
            count++;
        }
        prev = next;
        next = console.nextInt();
        // Point D
    }
    // Point E
    return count;
}
```

Categorize each assertion at each point below with either ALWAYS, NEVER or SOMETIMES.

	next == 0	prev == 0	next == prev
Point A			
Point B			
Point C			
Point D			
Point E			

Exercises

1. Write a sentinel loop that repeatedly prompts the user for names. Once an empty line is typed, all names typed are displayed. For example:

```
Type a person's name (or an empty line to stop): Marty
Type a person's name (or an empty line to stop): Andrea
Type a person's name (or an empty line to stop): Stuart
Type a person's name (or an empty line to stop):
Welcome to all: Marty Andrea Stuart
```

2. Write a sentinel loop that repeatedly prompts the user for numbers. Once any number less than zero is typed, the average of all non-negative numbers typed is displayed. Display the average as a double. For example:

```
Type a number: 7
Type a number: 4
Type a number: 16
Type a number: -4
Average was 9.0
```

If the first number typed is negative, do not print an average. For example:

```
Type a number: -2
```

3. Write code that prints a random number of lines between 2 and 10 lines inclusive, where each line contains a random number of 'x' characters between 5 and 19 inclusive. For example:

```
xxxxxxx
xxxxxxxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxxxx
xxxxxx
xxxxxxxxxxx
xxxxxxxxxxxxxxx
```

4. Write code to print between 5 and 10 random strings of letters (between 'a' and 'z'), one per line. Each string should have random length up to 80 characters.

5. Write a method named `makeGuesses` that will guess numbers between 1 and 50 inclusive until it makes a guess of at least 48. It should report each guess and at the end should report the total number of guesses made. Below is a sample execution:

```
guess = 43
guess = 47
guess = 45
guess = 27
guess = 49
total guesses = 5
```

6. Write interactive code that prompts for a desired sum, then repeatedly rolls two six-sided dice until their sum is the desired sum. Here is the expected dialogue with the user:

```
Desired dice sum: 9
4 and 3 = 7
3 and 5 = 8
5 and 6 = 11
5 and 6 = 11
1 and 5 = 6
6 and 3 = 9
```

7. Write a method named `gcd` that accepts two integers as parameters and returns the greatest common divisor of the two numbers. The greatest common divisor (GCD) of two integers a and b is the largest integer that is a factor of both a and b.

   One efficient way to compute the GCD of two numbers is to use Euclid's algorithm, which states the following:

   GCD(A, B) = GCD(B, A % B)
   GCD(A, 0) = Absolute value of A

8. Write a method named `showTwos` that shows the factors of 2 in a given integer. For example, the following calls:

```
showTwos(7);
showTwos(18);
showTwos(68);
showTwos(120);
```

should produce this output:

```
7 = 7
18 = 2 * 9
68 = 2 * 2 * 17
120 = 2 * 2 * 2 * 15
```

9. Write interactive that prompts for an integer and displays the same number in binary. For extra challenge, modify the program to make it display in another base, such as base 3 rather than base 2. The following is an example dialogue with the user:

```
Type a number: 44
44 in binary is 101100
```

10. Write a method named `randomWalk` that takes an integer n as a parameter and that performs n steps of a random one-dimensional walk, reporting the maximum position reached during the walk. The random walk should begin at position 0. On each step, you should either increase or decrease the position by 1 (each with equal probability). The output should look like this when the method is passed a parameter value of 7:

```
walking 7 steps
position = 1
position = 0
position = -1
position = 0
position = 1
position = 2
position = 1
max position = 2
```

11. Write a method named `season` that takes two integers as parameters representing a month and day and that returns a String indicating the season for that month and day. Assume that months are specified as an integer between 1 and 12 (1 for January, 2 for February, and so on) and that the day of the month is a number between 1 and 31.

    If the date falls between 12/16 and 3/15, you should return "Winter". If the date falls between 3/16 and 6/15, you should return "Spring". If the date falls between 6/16 and 9/15, you should return "Summer". And if the date falls between 9/16 and 12/15, you should return "Fall".

12. Write a method named `consecutive` that accepts three integers as parameters and returns true if they are three consecutive numbers; that is, if the numbers can be arranged into an order such that there is some integer k such that the parameters' values are k, k+1, and k+2. Your method should return false if the integers are not consecutive. Note that order is not significant; your method should return the same result for the same three integers passed in any order.

    For example, the calls `consecutive(1, 2, 3)`, `consecutive(3, 2, 4)`, and `consecutive(-10, -8, -9)` would return true. The calls `consecutive(3, 5, 7)`, `consecutive(1, 2, 2)`, and `consecutive(7, 7, 9)` would return false.

13. Write a method named `numUnique` that takes three integers as parameters and that returns the number of unique integers among the three. For example, the call `numUnique(18, 3, 4)` should return 3 because the parameters have 3 different values. By contrast, the call `numUnique(6, 7, 6)` would return 2 because there are only 2 unique numbers among the three parameters: 6 and 7.

Programming Projects

1. Write an interactive program that reads lines of input from the user and converts each line into "Pig Latin" language. Pig Latin is simply English but with the initial consonant sound moved to the end of each word, followed by "ay". Words that begin with vowels simply have an "-ay" appended. For example, the phrase:

   The deepest shade of mushroom blue

   would have the following appearance in Pig Latin:

   e-Thay eepest-day ade-shay of-ay ushroom-may ue-blay

   Terminate the program when the user types a blank line.

2. Write a reverse Hangman game where the user thinks of a word and the computer tries to guess the letters in that word. The user tells the computer how many letters are in their word.

3. Write a program that plays a guessing game with the user. The program should generate a random number between 1 and some maximum number such as 100, then prompt the user repeatedly to guess the number. When the user guesses incorrectly, the game gives the user a hint about whether the correct answer is higher or lower than their guess. Once the user guesses correctly, print a message showing the number of guesses used.

   Consider extending this program by making it play multiple games until the user selects to stop, and then printing statistics about the player's total and average number of guesses.

4. Write a program that plays a reverse guessing game with the user. The user thinks of a number between 1 and 10, and the computer repeatedly tries to guess it by guessing random numbers. It's fine for the computer to guess the same random number more than once. At the end of the game, the program reports how many guesses were needed.

```
This program has you, the user, choose a number
between 1 and 10, then I, the computer, will try
my best to guess it.

Is it 8? (y/n) n
Is it 7? (y/n) n
Is it 5? (y/n) n
Is it 1? (y/n) n
Is it 8? (y/n) n
Is it 1? (y/n) n
Is it 9? (y/n) y

I got your number of 9 correct in 7 guesses.
```

For added challenge, consider having the user hint to the computer whether the correct number is higher or lower than the computer's guess. The computer should adjust its range of random guesses based on the hint.

5. Write a game that plays many rounds of rock-paper-scissors. The user and computer will each choose between three items: rock (defeats scissors, but loses to paper), paper (defeats rock, but loses to scissors) and scissors (defeats paper, but loses to rock). If the player and computer choose the same item, the game is a tie.

   An extension of this program would be to write different algorithmic strategies for choosing the best item. Should the computer pick randomly? Should it always pick a particular item or repeating pattern of items? Should it count the number of times the opponent chooses various items and base its strategy on this history?

Stuart Reges
Marty Stepp

Chapter 6
File Processing

Copyright © 2006 by Stuart Reges and Marty Stepp

- 6.1 File Reading Basics
  - Data, Data Everywhere
  - File Basics
  - Reading a File with a Scanner
- 6.2 Details of Token-Based Processing
  - Structure of Files and Consuming Input
  - Scanner Parameters
  - Paths and Directories
  - A More Complex Input File
- 6.3 Line-Based Processing
  - String Scanners and Line/Token Combinations
- 6.4 Advanced File Processing
  - Output Files with PrintStream
  - Try/Catch Statements
- 6.5 Case Study: Weighted GPA

Introduction

In Chapter 4 we saw how to construct a `Scanner` object to read input from the console. Now we will see how to construct `Scanner` objects to read input from files. The idea is fairly straightforward, but Java does not make it easy to read from input files. This is unfortunate because many interesting problems can be formulated as file processing tasks. Many introductory computer science classes have abandoned file processing altogether or the topic has moved into the second course because it is considered too advanced for novices.

There is nothing intrinsically complex about file processing, but Java was not designed for file processing and Sun has not been particularly eager to provide a simple solution. Sun did, however, introduce the `Scanner` class as a way to simplify some of the details associated with reading files. The result is that file reading is still awkward in Java, but at least the level of detail is manageable.

Before we can write a file processing program, we have to explore some issues related to Java exceptions. Remember that exceptions are errors that halt the execution of a program. In the case of file processing, we might try to open a file that doesn't exist, which would generate an exception.

6.1 File Reading Basics

In this section we look at the most basic issues related to file processing. What are files and why do we care about them? What are the most basic techniques for reading files in a Java program? Once we have mastered these basics, we'll move on to a more detailed understanding of the different techniques we can use for processing files.

Data, Data Everywhere

We seem to have a fascination with data. When the field of Statistics emerged in the nineteenth century, there was an explosion of interest in gathering and interpreting large amounts of data. Mark Twain reported that the British statesman Benjamin Disraeli complained to him that, "There are three kinds of lies: lies, damn lies and statistics."

The advent of the internet has only added fuel to the fire. Today every person with an internet connection has access to a vast array of databases with information about every facet of our existence. Below are just a few examples:

- Go to www.landmark-project.com and click on the link for "Raw Data" and you will find data files about earthquakes, air pollution, baseball, labor, crime, financial markets, US history, geography, weather, national parks, a "world values survey" and more.
- At www.gutenberg.com you'll find thousands of online books including the complete works of Shakespeare and works by Sir Arthur Conan Doyle, Jane Austen, H. G. Wells, James Joyce, Albert Einstein, Mark Twain, Lewis Carroll, T.S. Eliot, Edgar Allan Poe and many others.
- There is a wealth of genomic data on the web available from sites like www.gdb.com and www.ncbi.nih.gov. Biologists have decided that the vast quantities of data about the human genome and the genomes of other organisms should be publicly available to everyone to study.
- Many popular web sites like the Internet Movie Database make their data available for download as simple data files (described at www.imdb.com/interfaces).
- The US government produces reams of statistical data. The web site www.fedstats.gov provides a lengthy list of available downloads including maps and statistics on employment, climate, manufacturing, demographics, health, crime and so on.

Did you Know: Origin of Data Processing

Data processing as a field predates computers by over half a century. It is often said that necessity is the mother of invention and we find a good example of this principle in the emergence of data processing. The crisis that spawned the industry came from a requirement included in the US Constitution. Article 1, Section 2 of the US Constitution indicates that population will be used to decide how many representatives each state will get in the House of Representatives. To make such a calculation, you need to know the population numbers. So the

Constitution says, "The actual Enumeration shall be made within three Years after the first Meeting of the Congress of the United States, and within every subsequent Term of ten Years, in such Manner as they shall by Law direct."

The first census was completed relatively quickly in 1790. Since then, every ten years the US government has had to perform another complete census of the population. This process became more and more difficult as the population of the country grew larger. By 1880 the government found that with old-fashioned hand-counting techniques, it barely completed the census within the ten years allotted to it. So the government announced a competition for inventors to propose machines that could be used to speed up the process.

Herman Hollerith won the competition with a system involving punched cards. Over 62 million cards were punched by clerks and then counted by a series of one-hundred counting machines. This allowed the tabulation to be completed in less than half the time taken to hand-count the 1880 results even though the population had increased by twenty-five percent.

Hollerith struggled for years to turn his invention into a commercial success. His biggest problem initially was that he had just one customer: the US government. Eventually he found other customers and the company that he founded merged with competitors and grew into the company we now know as International Business Machines Corporation or IBM.

We think of IBM as a computer company, but it sold a wide variety of data processing equipment involving Hollerith cards long before computers became popular. Later when it entered the computer field, IBM used Hollerith cards for storing programs and data. One of the authors of this book used Hollerith punched cards in his freshman computer programming class in 1978.

File Basics

When you store data like this on your own computer, you store it in a *file*.

File

A collection of information stored on a computer.

Every file has a name. For example, if you were to download the text of *Hamlet* from the Gutenberg site, you might store it on your computer in a file called hamlet.txt. File names often end with a special suffix that indicates the kind of data it contains or the format it has been stored in. This suffix is known as a *file extension*. Below is a table of common file extensions.

Common File Extensions

Extension	Description
.txt	text file
.dat	data file
.out	output file
.java	Java source code file
.class	compiled Java bytecode file
.doc	Microsoft Word file
.xls	Microsoft Excel file
.pdf	Portable Document File by Adobe
.csv	Comma Separated Values file

We sometimes refer to files stored on your computer as *external files* because they are not contained within a program and are not obtained from the user during program execution. To access such a file from inside a Java program, you need to construct some kind of internal object that will represent the file. The Java class libraries include a class called File that performs this duty.

You construct a File object by passing the name of a file, as in:

```
File f = new File("hamlet.txt");
```

Once constructed, you can call a number of methods to manipulate the file. For example, the program below calls a method that determines whether a file exists, whether it can be read and what it's length is (i.e., how many characters are in the file):

```
 1  // Report some basic information about a file
 2
 3  import java.io.*;
 4
 5  public class FileInfo {
 6      public static void main(String[] args) {
 7          File f = new File("hamlet.txt");
 8          System.out.println("exists returns " + f.exists());
 9          System.out.println("canRead returns " + f.canRead());
10          System.out.println("length returns " + f.length());
11      }
12  }
```

Notice that it includes an import from the package java.io because the File class is part of that package. The term "io" is a bit of jargon used by computer science professionals to mean "input/output". Assuming you have stored the file hamlet in the same directory as the program, you get the following output when you run this program:

```
exists returns true
canRead returns true
length returns 191734
```

Below is a list of useful methods for File objects.

Useful Methods of `File` Objects

Method	Description
`delete()`	deletes the given file
`exists()`	whether or not this file exists on the system
`getAbsolutePath()`	the full path where this file is located
`getName()`	the name of this file as a `String`, without any path attached
`getPath()`	the path where this file is located, without the actual file's name
`isDirectory()`	whether this file represents a directory/folder on the system
`isFile()`	whether this file represents a file (non-folder) on the system
`length()`	the number of characters in this file
`mkdirs()`	creates the directory represented by this file, if it does not exist
`renameTo(File)`	changes this file's name to be the given file's name

Reading a File with a Scanner

The `Scanner` class that we have been using since Chapter 4 is flexible in that we can attach `Scanner` objects to many different kinds of input. You can think of a `Scanner` object as being like a faucet that you can attach to a pipe that has water flowing through it. For example, in a house we'd attach a faucet to a pipe carrying water from the city or from a well. But we also see faucets in mobile homes and airplanes where the source of water is different.

We have been constructing our `Scanner` objects by passing `System.in` to the `Scanner` constructor:

```
Scanner console = new Scanner(System.in);
```

This instructs the computer to construct a `Scanner` that reads from the console (i.e, pausing for input from the user). Instead of passing `System.in` to the constructor, we can pass a File object:

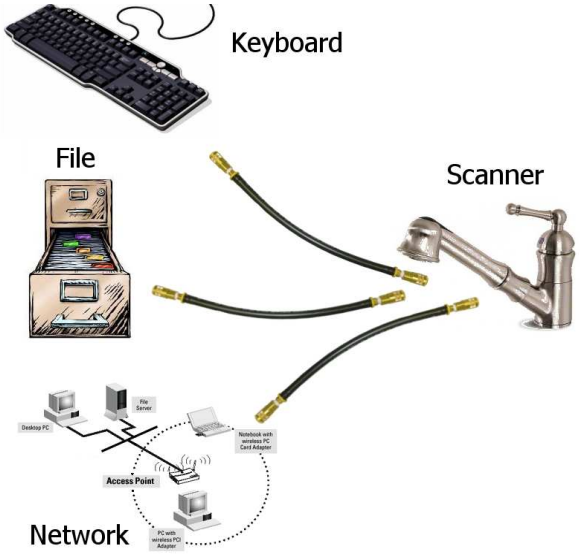
```
File f = new File("hamlet.txt");
Scanner input = new Scanner(f);
```

In this case, the variable f is not necessary, so we can shorten this to:

```
Scanner input = new Scanner(new File("hamlet.txt"));
```

This particular line of code or something like it will appear in all of your file processing programs.

`Scanner`s can be connected to many input sources.



Unfortunately, we run into a snag when we try to compile a program that constructs a Scanner in this manner. Suppose that we write a main method that begins by opening a Scanner in this manner:

```
// Flawed method--does not compile.
public static void main(String[] args) {
    Scanner input = new Scanner(new File("hamlet.txt"));
    ...
}
```

This program does not compile. It produces a message like the following:

```
Count.java:8: unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
        Scanner input = new Scanner(new File("hamlet.txt"));
                        ^
1 error
```

The issue involves exceptions, which were first introduced in Chapter 4. Remember that exceptions are errors that prevent a programming from continuing normal execution. In this case the compiler is worried that it might not be able to find a file called `hamlet.txt`. What is it supposed to do if that happens? It wouldn't have any way to continue executing the rest of the code because it wouldn't have a file to read from.

If the program is unable to locate the specified input file, it will generate an error by throwing what is known as a `FileNotFoundException`. This particular exception is known as a *checked exception*.

Checked Exception

An exception that *must* be caught or specifically declared in the header of the method that might generate it.

Because `FileNotFoundException` is a checked exception, we can't just ignore it. Java provides a construct known as the *`try/catch` statement* for handling such errors. Later in this chapter we will see how to use a `try/catch` to handle this error. But for now we will use a less sophisticated but

simpler approach. Java allows us to avoid handling this error as long as we clearly indicate the fact that we aren't handling the error. In particular, we can include what is known as a *throws clause* in the header for the `main` method to clearly state the fact that our `main` method might generate this exception.

> **`throws` clause**
>
> A declaration that a method will not attempt to handle a particular type of exception.

We can modify the header for method main to include a throws clause indicating that it can throw a FileNotFoundException:

```
public static void main(String[] args) throws FileNotFoundException {
    Scanner input = new Scanner(new File("hamlet.txt"));
    ...
}
```

With this modification, the program compiles. Once we've constructed the `Scanner` so that it reads from the file, we can manipulate it like any other `Scanner`. When we read from the console, we always prompt before reading to give the user an indication of what kind of data we want. When we read from a file, we don't need to prompt because the data is already there, stored in the file. For example, we might write the following program to count the number of words in Hamlet:

```
 1  // Counts the number of words in Hamlet.
 2
 3  import java.io.*;
 4  import java.util.*;
 5
 6  public class Count {
 7      public static void main(String[] args) throws FileNotFoundException {
 8          Scanner input = new Scanner(new File("hamlet.txt"));
 9          int count = 0;
10          while (input.hasNext()) {
11              String word = input.next();
12              count++;
13          }
14          System.out.println("total words = " + count);
15      }
16  }
```

We have to include an import from java.util for the Scanner class and an import from java.io for the File class. The program generates the following output:

```
total words = 31956
```

6.2 Details of Token-Based Processing

Now that we have seen some of the basic issues involved with reading an external input file, let's explore in more detail the issues involved in reading a file. One way to process a file is token by token.

Token-Based Processing

Processing input token by token (i.e., one word at a time or one number at a time).

Remember from Chapter 3 that the primary token-reading methods for the Scanner class are:

- nextInt() for reading an int value
- nextDouble() for reading a double value
- next() for reading the next token as a String

For example, we might want to create a file called `numbers.dat` with the following content.

```
308.2 14.9 7.4
2.8


3.9 4.7 -15.4
2.8
```

You can create such a file with an editor like NotePad on Windows or TextEdit on the Macintosh. Then you might write a program that processes this external input file and produces some kind of report. For example, the program below reads the first five numbers from the file and reports their sum.

```
 1  // Program that reads 5 numbers and reports their sum.
 2
 3  import java.io.*;
 4  import java.util.*;
 5
 6  public class Echo1 {
 7      public static void main(String[] args) throws FileNotFoundException {
 8          Scanner input = new Scanner(new File("numbers.dat"));
 9
10          double sum = 0.0;
11          for (int i = 1; i <= 5; i++) {
12              double next = input.nextDouble();
13              System.out.println("number " + i + " = " + next);
14              sum += next;
15          }
16          System.out.println("Sum = " + sum);
17      }
18  }
```

Remember that we need a throws clause in the header for main because of the potential FileNotFoundException. The program produces the following output:

```
number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
number 4 = 2.8
number 5 = 3.9
Sum = 337.19999999999993
```

Notice that the numbers are not reported as adding up to 337.2. This is another example of a round-off error, which was described in Chapter 4.

The preceding program reads exactly five numbers from the file. More typically we read indefinitely using a while loop as long as there are more numbers to read. Remember that the `Scanner` class includes a series of `hasNext` methods that parallel the various `next` methods. In this case, we are using `nextDouble` to read a value of type double, so we can use `hasNextDouble` to test whether there is such a value to read.

```
1  // Variation that reads while there are more numbers to read.
2
3  import java.io.*;
4  import java.util.*;
5
6  public class Echo2 {
7      public static void main(String[] args) throws FileNotFoundException {
8          Scanner input = new Scanner(new File("numbers.dat"));
9
10         double sum = 0.0;
11         int count = 0;
12         while (input.hasNextDouble()) {
13             double next = input.nextDouble();
14             count++;
15             System.out.println("number " + count + " = " + next);
16             sum += next;
17         }
18         System.out.println("Sum = " + sum);
19     }
20 }
```

This program would work on an input file with any number of numbers. Our file happens to have eight numbers and when we run this version of the program, we get the following output:

```
number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
number 4 = 2.8
number 5 = 3.9
number 6 = 4.7
number 7 = -15.4
number 8 = 2.8
Sum = 329.29999999999995
```

Structure of Files and Consuming Input

We think of text as being two-dimensional, like a sheet of paper, but from the computer's point of view, each file is just a one-dimensional sequence of characters. For example, consider the file called `numbers.dat` that we saw in the last section:

```
308.2 14.9 7.4
2.8


3.9 4.7 -15.4
2.8
```

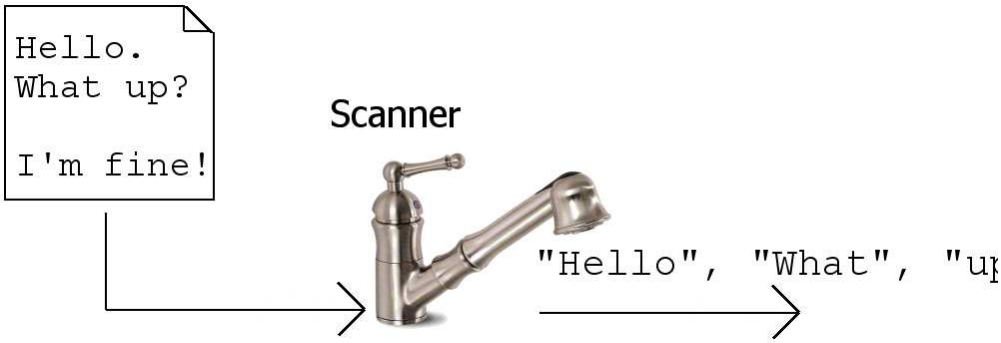
We think of this as a 6-line file with text going across and down and two blank lines in the middle. The computer views the file differently. When someone types in a file like this, they hit the Enter key to go to a new line. This inserts special "new line" characters in the file. We have seen that the escape sequence `\n` can be used to produce a newline character for output. We can annotate the file above with `\n` characters to indicate the end of each line:

```
308.2 14.9 7.4\n
2.8\n
\n
\n
3.9 4.7 -15.4\n
2.8\n
```

Once we have marked the end of each line, we no longer need to use a 2-dimensional representation. We can collapse this to a one-dimensional sequence of characters:

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
```

`Scanner`s treat files as one-dimensional strings of characters and convert their contents into a series of whitespace-separated tokens.



From this one-dimensional sequence, we can reconstruct the various lines of the file. This is how the computer views the file, as a one-dimensional sequence of characters including special characters that represent "new line". On some systems, including Windows machines, there are two different characters that represent "new line", but for our discussion, we'll use just `\n` to represent both. Objects like `Scanner` handle these differences for us so we can generally ignore them. (For those who are interested, the brief explanation is that Windows machines end each line with a `\r` followed by a `\n`.)

To process a file the `Scanner` object keeps track of a current position in the file. You can think of this as a cursor or pointer into the file.

Input cursor

A pointer to the current position in an input file.

When the `Scanner` object is first constructed, this cursor points to the beginning of the file. But as we perform various `next` operations, this cursor moves forward. The Echo2 program from the last section processes the file through a series of calls on `nextDouble`. Let's take a moment to examine in detail how that works. When the `Scanner` is first constructed, the input cursor will be positioned at the beginning of the file (indicated below with an up-arrow pointing at the first character in the file):

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
^
input
cursor
```

After the first call on `nextDouble`, the cursor will be positioned in the middle of the first line after the token `"308.2"`.

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
     ^
   input
   cursor
```

We refer to this process as consuming input.

Consuming input

Moving the input cursor forward past some input.

The first call on `nextDouble` consumes the text `"308.2"` from the input file and leaves the input cursor positioned at the first character after this token. Notice that this leaves the input cursor positioned at a space. When the second call is made on `nextDouble`, the `Scanner` first skips past this space to get to the next token and then consumes the text `"14.9"` and leaves the cursor positioned at the space that follows it:

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
         ^
        input
        cursor
```

A third call on `nextDouble` skips the space it is positioned at and consumes the text `"7.4"`.

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
            ^
           input
           cursor
```

At this point, the input cursor is positioned at the newline character at the end of the first line of input. A fourth call on `nextDouble` skips past this newline character and consumes the text `"2.8"`.

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                 ^
               input
               cursor
```

Notice that in skipping past the first newline character, the input cursor has moved into data stored on the second line of input (the 2.8). At this point, the input cursor is positioned at the end of the second line of input because it has consumed the `"2.8"` token. When a fifth call is made on `nextDouble`, the `Scanner` finds two newline characters in a row. This isn't a problem for the `Scanner`, because it simply skips past any leading whitespace characters (spaces, tabs, newline characters) until it finds an actual token. So it skips both of these newline characters and consumes the text `"3.9"`.

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                             ^
                          input
                          cursor
```

At this point the input cursor is positioned in the middle of the fifth line of input (the third and fourth lines were blank). The sixth call on `nextDouble` consumes the text `"4.7"`:

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                               ^
                            input
                            cursor
```

The seventh call consumes the text `"-15.4"`:

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                                    ^
                                 input
                                 cursor
```

And the eight call consumes the text `"2.8"`:

```
308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                                         ^
                                      input
                                      cursor
```

At this point the input cursor is positioned at the newline character at the end of the file. If you attempted to call `nextDouble` again, it would throw an exception because there are no more tokens left to process. But remember that the Echo2 program has a while loop that calls `hasNextDouble` before calling `nextDouble` to make sure that there actually is a double value to process. When the input cursor reaches the newline at the end of the file, the `Scanner` notices that there are no more double values to read and it returns false when `hasNextDouble` is called. That stops the while loop from executing and you exit the program.

When you call methods like `hasNextDouble`, the `Scanner` looks ahead in the file to see whether there is a next token and whether it could be interpreted as being of that type (in this case a double). So the Echo2 program will continue executing until it reaches the end of the file or until it encounters a token that could not be interpreted as a double.

Scanner objects are designed for file processing in a forward manner. They provide a great deal of flexibility for looking ahead in an input file, but no support for reading the input backwards. There are no "previous" methods and no mechanism for resetting a Scanner back to the beginning of the input. If a program needs to read an input file more than once, you would need to construct more than one Scanner object tied to the same file.

Scanner Parameters

Novices are sometimes surprised that the input cursor for a Scanner does not reset to the beginning of the file, particularly when it is passed as a parameter to a method. For example, consider the following variation of the Echo1 program. It has a method that takes a Scanner as input and an integer specifying how many numbers to process:

```
1  // Demonstrates a Scanner as a parameter to a method that
2  // can consume an arbitrary number of tokens
3
4  import java.io.*;
5  import java.util.*;
6
7  public class Echo3 {
8      public static void main(String[] args) throws FileNotFoundException {
9          Scanner input = new Scanner(new File("numbers.dat"));
10         processTokens(input, 2);
11         processTokens(input, 3);
12         processTokens(input, 2);
13     }
14
15     public static void processTokens(Scanner input, int n) {
16         double sum = 0.0;
17         for (int i = 1; i <= n; i++) {
18             double next = input.nextDouble();
19             System.out.println("number " + i + " = " + next);
20             sum += next;
21         }
22         System.out.println("Sum = " + sum);
23         System.out.println();
24     }
25 }
```

The main method creates a Scanner object that is tied to the numbers.dat file. It then calls the processTokens method several times indicating the number of tokens to process. The first call instructs the method to process 2 tokens, which operates on the first two tokens of the file, generating the following output:

```
number 1 = 308.2
number 2 = 14.9
Sum = 323.09999999999997
```

What do you suppose the second call on the method does? The second call indicates that 3 tokens are to be processed. Some people expect the method to process the first three tokens of the file, but that's not what happens. Remember that the Scanner keeps track of where the input cursor is positioned. After the first call on the method, the input cursor is positioned beyond the first two tokens. So the second call on the method processes the next three tokens from the file, producing this output:

```
number 1 = 7.4
number 2 = 2.8
number 3 = 3.9
Sum = 14.1
```

The final call on the method asks the Scanner to process the next two tokens from the file, which ends up processing the sixth and seventh numbers from the file:

```
number 1 = 4.7
number 2 = -15.4
Sum = -10.7
```

The program then terminates, never having processed the eighth number in the file.

The key point to remember is that a Scanner keeps track of the input cursor so that you can process an input file piece by piece in a very flexible manner. Even when the Scanner is passed as a parameter to a method, it remembers how much of the input file has been processed so far.

Paths and Directories

Any given file will be stored in a particular folder or directory. Directories are organized in a hierarchy starting from a root directory at the top. For example, most Windows machines have a disk drive known as C:. At the top level of this drive, we have what is known as the *root* directory, which we can describe as C:\. There will be various top-level directories contained in this root directory and each of them can have subdirectories and each subdirectory can have subdirectories and so on. Every file is stored in one of these directories. The description of how to get from the top level directory to the particular directory that stores a file is known as the *path* of the file.

File Path

A description of a file's location on a computer starting with a drive and including the path from the root directory to the directory where the file is stored.

For example, a file might be stored in C:\school\data\hamlet.txt. We read the path information from left to right. The file is on the C: drive in a folder called school and in a subfolder called data.

In the previous section we used the file name `numbers.dat`. When Java finds you using a simple name like that (also called a *relative* file path), it looks in the *current directory* to find the file.

Current directory (also called "working directory")

The directory that Java uses as the default when a simple file name is used.

The definition of *current directory* varies depending upon what Java environment you are using. In most environments, the current directory is the directory in which your program appears. This is the behavior we'll assume for the examples in this textbook.

You can also use a fully-qualified file name (sometimes called an *absolute file path*). This approach works well only when you know exactly where your file is going to be stored on your system. For example, if you are on a Windows machine and you have stored the file in a directory known as c:\data, we could use a file name like this:

```
Scanner input = new Scanner(new File("c:/data/numbers.dat"));
```

Notice that the path is written with forward slash characters rather than backslash characters. On Windows you would normally use a backslash, but Java allows you to use a forward slash instead. If we wanted to use a backslash, we would have to use an escape sequence. Most programmers use the simpler approach of using forward slash characters because Java does the appropriate translation on Windows machines.

Sometimes rather than writing a file's path name in the code ourselves, we ask the user for a file name. In the last chapter we saw a program called FindSum that prompted the user for a series of numbers to add together. Below is a variation that prompts the user for the name of a file of numbers to be added together.

```
1  // This program adds together a series of numbers from a file.  It prompts
2  // the user for the file name, then reads the file and reports the sum.
3
4  import java.io.*;
5  import java.util.*;
6
7  public class FindSum2 {
8      public static void main(String[] args) throws FileNotFoundException {
9          System.out.println("This program will add together a series of real");
10         System.out.println("numbers from a file.");
11         System.out.println();
12
13         Scanner console = new Scanner(System.in);
14
15         System.out.print("What is the file name? ");
16         String name = console.nextLine();
17
18         Scanner input = new Scanner(new File(name));
19         System.out.println();
20
21         double sum = 0;
22         while (input.hasNextDouble()) {
23             double next = input.nextDouble();
24             sum += next;
25         }
26         System.out.println("Sum = " + sum);
27     }
28 }
```

We read the file name using a call on `nextLine` to read an entire line of input from the user. This allows the user to type in file names that have spaces in them. Notice that we still need the `throws FileNotFoundException` in the header for `main` because even though we are prompting the user for a file name, there won't necessarily be a file of that name.

If we have this program read from the file `numbers.dat` that we saw in the last section, then the program would execute like this:

```
This program will add together a series of real
numbers from a file.

What is the file name? numbers.dat

Sum = 329.29999999999995
```

The user also has the option of specifying a full file name, as in:

```
This program will add together a series of real
numbers from a file.

What is the file name? c:\data\numbers.dat

Sum = 329.29999999999995
```

Notice that the user doesn't have to type two backslashes to get a single backslash. That's because the `Scanner` object that reads the user's input is able to read it without escape sequences.

A More Complex Input File

Suppose that you have an input file that has information about how many hours have been worked by each employee of a company. For example, it might look like the following:

```
Erica 7.5 8.5 10.25 8 8.5
Greenlee 10.5 11.5 12 11 10.75
Simone 8 8 8
Ryan 6.5 8 9.25 8
Kendall 2.5 3
```

The idea is that we have a list of hours worked by each employee and we want to find out the total hours worked by each individual. We can construct a `Scanner` object linked to this file to solve this task. As you start writing more complex file processing programs, you will want to divide the program into methods to break up the code into logical subtasks. In this case, we can open the file in `main` and write a separate method to process the file.

Most file processing will involve while loops because we won't know in advance how much data the file has in it. We'll choose different tests depending upon the particular file we are processing, but they will almost all be calls on the various `hasNext` methods of the `Scanner` class. We basically want to say, "while you have more data for me to process, let's keep reading."

In this case we have a series of input lines that each begin with a name. For this program we are assuming that names are simple, with no spaces in the middle. That means we'll be reading them with a call on the `next` method. As a result, our overall test involves seeing if there is another name in the input file:

```
while (input.hasNext()) {
    <process next person>
}
```

So how do we process one person? We have to read their name and then read their list of hours. If you look at the sample input file, you will see that the list of hours is not always the same length. This is a common occurrence in input files. For example, some employees might have worked on 5 different days while others worked only 2 days or 3 days. So we will use a loop for this as well. This is a nested loop. The outer loop is handling one person at a time and the inner loop will handle one number at a time. The task is a fairly straightforward cumulative sum:

```
double sum = 0.0;
while (input.hasNextDouble()) {
    sum += input.nextDouble();
}
```

Putting this all together, we end up with the following complete program.

```
 1  // This program reads an input file of hours worked by various employees.  Each
 2  // line of the input file should have an employee's name (without any spaces)
 3  // followed by a list of hours worked, as in:
 4  //
 5  //     Erica 7.5 8.5 10.25 8
 6  //     Greenlee 10.5 11.5 12 11
 7  //     Ryan 6.5 8 9.25 8
 8  //
 9  // The program reports the total hours worked by each employee.
10
11  import java.io.*;
12  import java.util.*;
13
14  public class HoursWorked {
15      public static void main(String[] args) throws FileNotFoundException {
16          Scanner input = new Scanner(new File("hours.dat"));
17          process(input);
18      }
19
20      public static void process(Scanner input) {
21          while (input.hasNext()) {
22              String name = input.next();
23              double sum = 0.0;
24              while (input.hasNextDouble()) {
25                  sum += input.nextDouble();
26              }
27              System.out.println("Total hours worked by " + name + " = " + sum);
28          }
29      }
30  }
```

Notice that we again need the `throws FileNotFoundException` in the header for `main`. We don't need to include this in the `process` method because the code to open the file appears in method `main`.

If we put the input above into a file called `hours.dat` and execute the program, we get the following result.

```
Total hours worked by Erica = 42.75
Total hours worked by Erin = 55.75
Total hours worked by Simone = 24.0
Total hours worked by Ryan = 31.75
Total hours worked by Kendall = 5.5
```

6.3 Line-based Processing

So far we have been looking at programs that process input token by token. We often find ourselves working with input files that are line-based where each line of input represents a different case to be handled separately from the rest. This leads to a second style of file processing.

Line-Based Processing

Processing input line by line (i.e., reading in entire lines of input at a time).

Most file processing involves a combination of these two styles and the `Scanner` class is flexible enough to allow us to write programs that have both styles of processing. For line-based processing, we use the `nextLine` and `hasNextLine` methods of the `Scanner` object. For example, below is a program that echos an input file in uppercase:

```
1  // Reads a file and echos it in uppercase.
2
3  import java.io.*;
4  import java.util.*;
5
6  public class EchoUppercase {
7      public static void main(String[] args) throws FileNotFoundException {
8          Scanner input = new Scanner(new File("poem.txt"));
9          while (input.hasNextLine()) {
10             String text = input.nextLine();
11             System.out.println(text.toUpperCase());
12         }
13     }
14 }
```

This loop reads the input line by line until it runs out of lines to process, printing each line in uppercase. It reads from a file called poem.txt. Given this input file:

```
My candle burns at both ends
It will not last the night;
But ah, my foes, and oh, my friends -
It gives a lovely light.

          --Edna St. Vincent Millay
```

It produces the following output:

```
MY CANDLE BURNS AT BOTH ENDS
IT WILL NOT LAST THE NIGHT;
BUT AH, MY FOES, AND OH, MY FRIENDS -
IT GIVES A LOVELY LIGHT.

          --EDNA ST. VINCENT MILLAY
```

Notice that we could not have accomplished the same task with token-based processing. Here the line breaks are significant because they are part of the poem. Also, when we read a file token by token, we lose the spacing within the line because the Scanner skips any leading whitespace when it reads a token. The final line of this input file is indented to make it clear that it is the name of the author and not part of the poem. We would lose that spacing if we read the file as a series of tokens.

String Scanners and Line/Token Combinations

In the last section we looked at a program called HoursWorked that processed the following data file:

```
Erica 7.5 8.5 10.25 8 8.5
Greenlee 10.5 11.5 12 11 10.75
Simone 8 8 8
Ryan 6.5 8 9.25 8
Kendall 2.5 3
```

In this case, the data is line-oriented with one employee's information on a different line of input, but we didn't incorporate this aspect of the data into our program. We processed the file in a token-based manner. In a sense, we got lucky that this worked out. For example, consider a slight variation in the data where each line of input begins with an employee id and a name rather than just a name:

```
101 Erica 7.5 8.5 10.25 8 8.5
783 Erin 10.5 11.5 12 11 10.75
114 Simone 8 8 8
238 Ryan 6.5 8 9.25 8
156 Kendall 2.5 3
```

This seems like a fairly simple change that shouldn't require major changes to the code. Recall that the program has a method to process the file:

```
public static void process(Scanner input) {
    while (input.hasNext()) {
        String name = input.next();
        double sum = 0.0;
        while (input.hasNextDouble()) {
            sum += input.nextDouble();
        }
        System.out.println("Total hours worked by " + name + " = " + sum);
    }
}
```

Suppose we add a line of code to read the employee id and we modify the println to report it:

```
public static void process(Scanner input) {
    while (input.hasNext()) {
        int id = input.nextInt();
        String name = input.next();
        double sum = 0.0;
        while (input.hasNextDouble()) {
            sum += input.nextDouble();
        }
        System.out.println("Total hours worked by " + name +
                           " (id#" + id + ") = " + sum);
    }
}
```

When we run this version of the program on the new input file, we get one line of output and then an exception is thrown:

```
Total hours worked by Erica (id#101) = 825.75
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        ...
```

The program correctly reads Erica Kane's employee id and reports it in the println statement, but then it dies. Also, notice that the program reports Erica's total hours worked as 825.75 when it should be 42.75. Where did it go wrong?

If you compute the difference between the reported sum of 825.75 hours and the correct sum of 42.75 hours, you'll find it is equal to 783. That number appears in our data file. It's the employee id of the second employee, Erin. In adding up the hours for Erica, the program has accidentally read Erin's employee id and added it to the sum. That's also why the exception is happening because on the second iteration of the loop, we are trying to read an employee id for Erin when the next token in the file is not an integer.

The solution is to somehow get the program to stop reading when it gets to the end of an input line. Unfortunately, there is no easy way to do this with a token-based approach. Our loop is asking whether the Scanner has a next double value to read and the employee id will look like a double that can be read. We might try to form a complex test that looks for a double that is not also an integer, but even that won't work because some of the hours are integers.

To make this program work, we need to write a more sophisticated program that pays attention to the line breaks. We have to read an entire line of input at a time and process that line by itself. Recall that the `main` method for the program looks like this:

```
public static void main(String[] args) throws FileNotFoundException {
    Scanner input = new Scanner(new File("hours2.dat"));
    process(input);
}
```

If we incorporate a line-based loop, we end up with something like this:

```
public static void main(String[] args) throws FileNotFoundException {
    Scanner input = new Scanner(new File("hours2.dat"));
    while (input.hasNextLine()) {
        String text = input.nextLine();
        processLine(text);
    }
}
```

By reading the file line by line, we guarantee that we don't accidentally combine data for two employees. The downside is that we have to write a method called processLine that takes a String as a parameter and we have to pull apart that String. It contains the employee id followed by the employee name followed by the numbers indicating how many hours were worked on different days. In other words, the input line is composed of several pieces (several tokens) that we want to process piece by piece. It was much easier to process this data in a token-based manner than to have it in a String.

Fortunately there is a convenient way to do this. We can construct a `Scanner` object from an individual `String`. Remember that just as a faucet can be attached to different sources of water (a faucet in a house attached to city water or well water versus a faucet on an airplane attached to a tank of water), we can attach a `Scanner` to different sources of input. We've seen how to attach it to the console (`System.in`) and to a file (passing a `File` object). We can also attach it to an individual `String`. For example, we might write this code:

```
Scanner input = new Scanner("18.4 17.9 8.3 2.9");
```

This constructs a `Scanner` that gets its input from the `String` that we used to construct it. This `Scanner` has an input cursor just like a `Scanner` linked to a file. Initially the input cursor is positioned at the first character in the `String` and it moves forward as we read tokens from the `Scanner`.

Below is a short program to demonstrate this:

```
 1  import java.util.*;
 2
 3  public class StringScannerExample {
 4      public static void main(String[] args) {
 5          Scanner input = new Scanner("18.4 17.9 8.3 2.9");
 6          while (input.hasNextDouble()) {
 7              double next = input.nextDouble();
 8              System.out.println(next);
 9          }
10      }
11  }
```

It produces the following output:

```
18.4
17.9
8.3
2.9
```

Notice that it produces four lines of output because there are four numbers in the `String` that we use to construct the `Scanner`.

When we have a file that requires this combination of line-based processing and token-based processing, we can use a slightly different approach by constructing a different `String`-based `Scanner` for each line of the input file. With this approach, we end up with a lot of `Scanner` objects. We have a `Scanner` object that is keeping track of the external input file. We use that `Scanner` to read entire lines of input. But each time we read a line of text from the file, we construct a mini-Scanner for just that line of input. We can then used token-based processing for these mini-Scanner objects because they each contain just a single line of data in them.

This combination of line-based and token-based processing is powerful. You will find that the approach (and slight variations) can be used to process a large variety of input files.

In the case of the HoursWorked program, each input line contained information for a single employee. Processing the input line involves making a Scanner for the line and then reading its various parts in a token-based manner (id, name, hours). We can put this all together into a new version of the program:

```
1  // This program reads an input file of hours worked by various employees.  Each
2  // line of the input file should have an employee's name (without any spaces)
3  // followed by a list of hours worked, as in:
4  //
5  //     Erica 7.5 8.5 10.25 8
6  //     Greenlee 10.5 11.5 12 11
7  //     Ryan 6.5 8 9.25 8
8  //
9  // The program reports the total hours worked by each employee.
10
11 import java.io.*;
12 import java.util.*;
13
14 public class HoursWorked2 {
15     public static void main(String[] args) throws FileNotFoundException {
16         Scanner input = new Scanner(new File("hours2.dat"));
17         while (input.hasNextLine()) {
18             String text = input.nextLine();
19             processLine(text);
20         }
21     }
22
23     // Processes the given String (id, name and hours worked)
24     public static void processLine(String text) {
25         Scanner data = new Scanner(text);
26         int id = data.nextInt();
27         String name = data.next();
28         double sum = 0.0;
29         while (data.hasNextDouble()) {
30             sum += data.nextDouble();
31         }
32         System.out.println("Total hours worked by " + name +
33                            " (id#" + id + ") = " + sum);
34     }
35 }
```

Notice that the `main` method includes the line-based processing of reading entire lines of input from the file. Each such line is passed to the processLine method. Each time we call processLine, we make a mini-Scanner for just that line of input and we use token-based processing (calling the methods `nextInt`, `next` and `nextDouble`).

This new version of the program produces the following output:

```
Total hours worked by Erica (id#101) = 42.75
Total hours worked by Erin (id#783) = 55.75
Total hours worked by Simone (id#114) = 24.0
Total hours worked by Ryan (id#238) = 31.75
Total hours worked by Kendall (id#156) = 5.5
```

While this version of the program is a little more complex than the original, it is much more flexible because it pays attention to line breaks.

6.4 Advanced File Processing

In this section we explore two advanced topics related to file processing. First we see how to produce external output files. Then we look at how to handle errors using try/catch statements.

Output Files with PrintStream

All of our programs so far have sent their output to the console window by calling `System.out.print` or `System.out.println`. Just as we can read input from an external file instead of reading from the console, we can write output to an external file instead of writing it to the console. There are many ways to accomplish this. The simplest approach is to take advantage of what you already know. By now, you've learned all about how `print` and `println` statements work. We can leverage that knowledge to allow you to easily create external output files.

If you look at Sun's Java documentation, you will find that `System.out` is a variable that stores a reference to an object of type `PrintStream`. The `print` and `println` statements you've been writing are calls on methods that are part of the `PrintStream` class. The variable `System.out` stores a reference to a special `PrintStream` object that is tied to the console window. But we can construct other `PrintStream` objects that send their output to other places. Suppose, for example, that we want to send output to a file called results.txt. We can construct a `PrintStream` object as follows:

```
PrintStream output = new PrintStream(new File("results.txt"));
```

This looks a lot like the line of code we've been using to construct a `Scanner` tied to an external input file. In this case, the computer is creating an external output file. If there is no such file, then the program creates it. If there is such a file, it overwrites the current version. Initially the file will be empty. It will end up containing whatever output you tell it to produce through calls on `print` and `println`.

This line of code can generate an exception if Java is unable to create the file you've described. There are many reasons this might happen. You might not have write access to the directory or the file might be locked in some way because it is being used by another program. Just as with the line of code that creates a file-based `Scanner`, this line of code potentially throws a `FileNotFoundException`. That means that Java will require us to have the `throws` clause in whatever method contains this line of code. The simplest approach is to have this line of code in `main`. In fact, it is common practice to have the `main` method begin with the lines of code that deal with these external files (both input and output).

Once you have constructed a `PrintStream` object, how do you use it? You already have a good idea of what to do. We have been making calls on System.out.print and System.out.println since Chapter 1. As it turns out, System.out is the name of a variable that stores a reference to a PrintStream object. So you have had quite a bit of practice already talking to the PrintStream known as System.out. Our plan is to have a variable called output that will store a reference to our PrintStream object. If you just think of everything that you know about System.out, you'll have a good idea of what to do. But for this program, you will call output.print instead of System.out.print and output.println instead of System.out.println.

As a simple example, remember that in Chapter 1, we looked at the following variation of the simple "hello world" program that produces several lines of output:

```
1  public class Hello3 {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4          System.out.println();
5          System.out.println("This program produces three lines of output.");
6      }
7  }
```

Below is a variation that sends its output to a file called `hello.txt`:

```
1  import java.io.*;
2  import java.util.*;
3
4  public class Hello4 {
5      public static void main(String[] args) throws FileNotFoundException {
6          PrintStream output = new PrintStream(new File("hello.txt"));
7          output.println("Hello world.");
8          output.println();
9          output.println("This program produces three lines of output.");
10     }
11 }
```

When you run this new version of the program, a curious thing happens. The program doesn't seem to do anything; no output appears on the console at all. That's because we're used to having the output of the program go to the console window. In this case, the output was directed to a file instead. After the program finishes executing, we can open up the file called `hello.txt` and we'll find that it contains the following:

```
Hello world.

This program produces three lines of output.
```

The main point is that everything you've learned to do with `System.out` you can also do with these `PrintStream` objects that are tied to files.

We can also write methods that take `PrintStream` objects as parameters. For example, consider the task of fixing the spacing for a series of words. We might have a line of text that has erratic spacing, as in:

```
a       new  nation,     conceived  in    liberty
```

Suppose we want to print this text with the spacing fixed so that there is exactly one space between each pair of words:

```
a new nation, conceived in liberty
```

How do we do that? Assume that we are writing a method that is passed a `String` to echo and a `PrintStream` object to send the output to:

```
public static void echoFixed(String text, PrintStream output)
```

We can construct a `Scanner` from the `String` and then use the `next` method to read one word at a time. Recall that the `Scanner` class ignores whitespace, so we'll get just the individual words without all of the spaces between them. As we read words, we need to echo them to the `PrintStream` object. Here's a first attempt:

```
Scanner data = new Scanner(text);
while (data.hasNext()) {
    output.print(data.next());
}
```

This code does a great job of deleting the long sequences of spaces from the `String`, but it goes too far. It eliminates all of the spaces. We want one space between each pair of words, so we have to include some spaces for our output:

```
Scanner data = new Scanner(text);
while (data.hasNext()) {
    output.print(data.next() + " ");
}
```

This ends up looking pretty good, but it prints an extra space at the end of the line. If we want to get rid of that space so that we truly have spaces appearing only between pairs of words, we have to change this slightly. This is a classic fencepost problem. We want the spaces between the words, so we have one more word than we have spaces. We can use the typical solution of processing the first word before the loop begins and turning around the order of the other two operations inside the loop (print a space and then the word):

```
Scanner data = new Scanner(text);
output.print(data.next());
while (data.hasNext()) {
    output.print(" " + data.next());
}
```

This now works well for almost all cases. By including our fencepost solution of echoing the first word before the loop begins, we've introduced an assumption that there is a first word. If the `String` has no words at all, then this call on `next` will throw an exception. So we need a test for the case where the `String` has no words at all. And if we also want this to produce a complete line of output, we have to include a call on `println` to complete the line of output after printing the individual words. Putting all of this together, we get the following:

```
public static void echoFixed(String text, PrintStream output) {
    Scanner data = new Scanner(text);
    if (data.hasNext()) {
        output.print(data.next());
        while (data.hasNext()) {
            output.print(" " + data.next());
        }
    }
    output.println();
}
```

Notice how we are now calling output.print and output.println instead of calling `System.out.print` and `System.out.println`. An interesting thing about this method is that it can be used to send output to an external output file but it can also be used to send output to `System.out`. The method header indicates that it works on any `PrintStream`. So we can call it to send output to a `PrintStream` object tied to an external file or we can call it to send output to `System.out`.

Below is a complete program that uses this method to fix the spacing in an entire input file of text. To underscore the flexibility of the method, the program sends its output to both the external file and the console. So you'll see the output appear in the console window as usual, but you'll also find that a file has been constructed called `words2.txt` that contains the output.

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class FixSpacing {
 5      public static void main(String[] args) throws FileNotFoundException {
 6          Scanner input = new Scanner(new File("words.txt"));
 7          PrintStream output = new PrintStream(new File("words2.txt"));
 8          while (input.hasNextLine()) {
 9              String text = input.nextLine();
10              echoFixed(text, output);
11              echoFixed(text, System.out);
12          }
13      }
14
15      public static void echoFixed(String text, PrintStream output) {
16          Scanner data = new Scanner(text);
17          if (data.hasNext()) {
18              output.print(data.next());
19              while (data.hasNext()) {
20                  output.print(" " + data.next());
21              }
22          }
23          output.println();
24      }
25  }
```

Given the following input file:

```
    four         score    and
seven    years ago    our
   fathers brought  forth on this     continent
a       new  nation,     conceived  in    liberty
  and  dedicated   to  the  proposition    that
   all   men    are     created    equal
```

It produces the following output file called `words2.txt`:

```
four score and
seven years ago our
fathers brought forth on this continent
a new nation, conceived in liberty
and dedicated to the proposition that
all men are created equal
```

Try/Catch Statements

Including the clause `throws FileNotFoundException` in the header for `main` allows our programs to compile, but it's not a very satisfying solution to the underlying problem. To actually handle the potential error, we'd want to use something called a `try/catch` statement. We will not be exploring all of the details of `try/catch`, but we will examine how to write some basic `try/catch` statements that we could use for file processing.

The general syntax of the `try/catch` statement is the following:

```
try {
    <statement>;
    <statement>;
        ...
    <statement>;
} catch (<type> <name>) {
    <statement>;
    <statement>;
        ...
    <statement>;
}
```

Notice that it is divided into two blocks using the keywords `try` and `catch`. The first block has the code you want to execute. The second block has error recovery code that should be executed if an exception is thrown. So think of this as saying, "Try to execute these statements, but if something goes wrong, I'm going to give you some other code in the `catch` part that you should execute if an error occurs."

Notice that the catch part of this statement has a set of parentheses in which you include a type and name. The type should be the type of exception you are trying to catch. The name can be any legal identifier. For example, in the case of our `Scanner` code, we know that a `FileNotFoundException` might be thrown. What do we do if the exception occurs? That's a tough question, but for now let's just write an error message.

```
try {
    Scanner input = new Scanner(new File("numbers.dat"));
} catch (FileNotFoundException e) {
    System.out.println("File not found");
}
```

This code says to try constructing the `Scanner` from the file `numbers.dat` but if the file is not found, then print an error message instead. This is the basic idea we want to follow, but there are several issues we must address to make this code work for us. First of all, there is a scope issue. The variable input isn't going to be much use to us if it's trapped inside the try block. So we have to declare the `Scanner` variable outside the `try/catch` statement:

```
Scanner input;
try {
    input = new Scanner(new File("numbers.dat"));
} catch (FileNotFoundException e) {
    System.out.println("File not found");
}
```

We have a bigger problem in that simply printing an error message isn't a good way to recover from this problem. How is the program supposed to proceed with execution if it can't read from the file? It probably can't. So what would be a more appropriate way to recover from the error? That depends a lot on the particular program you are writing, so the answer is likely to vary from one program to the next.

Let's explore how you might handle this when you are prompting the user for a file name in the console window. In that case, we could keep prompting the user until they give us a legal file name. Let's begin by modifying the code above to prompt and read a file name.

```
Scanner input;
System.out.print("What is the name of the input file? ");
String name = console.nextLine();
try {
    input = new Scanner(new File(name));
} catch (FileNotFoundException e) {
    System.out.println("File not found");
}
```

This code catches the potential exception and prints an error message, but we want to add a loop that executes while the user has not given us a legal file name. We want it to look something like this:

```
Scanner input;
while (user hasn't given a legal name) {
    <prompt for name>
    <try to open file, generating error message if illegal>
}
```

We have a classic problem of how to prime this loop so that it enters the first time through. We're trying to construct a `Scanner` from a file. When we succeed, we'll be giving a value to the variable `input`. Can we initialize input to something that would indicate that we aren't yet done? The answer is yes. There is a special keyword in Java called `null` that is used to represent "no object". We can initialize the variable input to null as a way to say, "This variable doesn't yet point to an actual object." The primary advantage of initializing the variable to null is that we can test whether it's null in the while loop.

```
Scanner input = null;
while (input == null) {
    <prompt for name>
    <try to open file, generating error message if illegal>
}
```

null

A Java keyword signifying the absence of an object. Assigning an object variable to `null` leaves it in an unusable state until a non-`null` value is later stored there.

We start the variable with the value null, so it enters the while loop the first time through. If the code in the `try/catch` fails to properly open the file, then the variable will still be null and we'll execute the loop a second time, prompting for another file name and trying to open it. If the code in the `try/catch` fails again, then we generate yet another error message and go through the loop a third time.

We can combine this pseudocode with the `try/catch` code we saw earlier. It seems prudent to modify the error message to make it clear that the user is being given another chance to enter a legal file name.

```
Scanner input = null;
while (input == null) {
    System.out.print("What is the name of the input file? ");
    String name = console.nextLine();
    try {
        input = new Scanner(new File(name));
    } catch (FileNotFoundException e) {
        System.out.println("File not found.  Please try again.");
    }
}
```

This loop executes repeatedly until the call on `new Scanner` inside the try block succeeds and gives the variable input a non-null value. This code could be included in method `main`, although we'd have to construct a `Scanner` for console input to be able to prompt the user for a file name. Below is a variation of the HoursWorked2 program that prompts for a file name.

```
 1  // Variation of HoursWorked2 that prompts for a file name
 2
 3  import java.io.*;
 4  import java.util.*;
 5
 6  public class HoursWorked3 {
 7      public static void main(String[] args) {
 8          Scanner console = new Scanner(System.in);
 9          Scanner input = null;
10          while (input == null) {
11              System.out.print("What is the name of the input file? ");
12              String name = console.nextLine();
13              try {
14                  input = new Scanner(new File(name));
15              } catch (FileNotFoundException e) {
16                  System.out.println("File not found.  Please try again.");
17              }
18          }
19          while (input.hasNextLine()) {
20              String text = input.nextLine();
21              processLine(text);
22          }
23      }
24
25      // Processes the given String (id, name and hours worked)
26      public static void processLine(String text) {
27          Scanner data = new Scanner(text);
28          int id = data.nextInt();
29          String name = data.next();
30          double sum = 0.0;
31          while (data.hasNextDouble()) {
```

```
32              sum += data.nextDouble();
33          }
34          System.out.println("Total hours worked by " + name +
35                             " (id#" + id + ") = " + sum);
36      }
37  }
```

Notice that we no longer need the `throws FileNotFoundException` in the header for `main` because we handle the potential exception. Here is a log of execution showing what happens when the user types in illegal file names:

```
What is the name of the input file? ours2.dat
File not found.  Please try again.
What is the name of the input file? hours2.txt
File not found.  Please try again.
What is the name of the input file? data.txt
File not found.  Please try again.
What is the name of the input file? file.dat
File not found.  Please try again.
What is the name of the input file? hours2.dat
Total hours worked by Erica (id#101) = 42.75
Total hours worked by Erin (id#783) = 55.75
Total hours worked by Simone (id#114) = 24.0
Total hours worked by Ryan (id#238) = 31.75
Total hours worked by Kendall (id#156) = 5.5
```

This code for opening a file is complicated enough that you might want to put it in its own static method. Below is a final variation that includes a method called getInput that prompts the user for a legal file name that can be used to construct a `Scanner`.

```
1  // Variation of HoursWorked3 with file opening code in a method
2
3  import java.io.*;
4  import java.util.*;
5
6  public class HoursWorked4 {
7      public static void main(String[] args) {
8          Scanner console = new Scanner(System.in);
9          Scanner input = getInput(console);
10         while (input.hasNextLine()) {
11             String text = input.nextLine();
12             processLine(text);
13         }
14     }
15
16     // prompt the user for a legal file name, create and return
17     // a Scanner tied to the file
18     public static Scanner getInput(Scanner console) {
19         Scanner result = null;
20         while (result == null) {
21             System.out.print("What is the name of the input file? ");
22             String name = console.nextLine();
23             try {
24                 result = new Scanner(new File(name));
25             } catch (FileNotFoundException e) {
26                 System.out.println("File not found.  Please try again.");
27             }
28         }
29         return result;
30     }
31
32     // Processes the given String (id, name and hours worked)
33     public static void processLine(String text) {
34         Scanner data = new Scanner(text);
35         int id = data.nextInt();
36         String name = data.next();
37         double sum = 0.0;
38         while (data.hasNextDouble()) {
39             sum += data.nextDouble();
40         }
41         System.out.println("Total hours worked by " + name +
42                            " (id#" + id + ") = " + sum);
43     }
44 }
```

The code we have written for opening a file tends to be fairly standard in that we could use it without modification in many programs. We refer to this as *boilerplate* code.

Boilerplate Code

Code that tends to be the same from one program to another.

The method getInput is a good example of the kind of boilerplate code that you might use in many different file-processing programs.

6.5 Case Study: Weighted GPA

The HoursWorked program required that names have no spaces in them. This isn't a very practical restriction. It would be more convenient to be able to type anything for a name, including spaces. One way to do that is to put the name on a separate line from the rest of the data. For example, suppose that you want to compute weighted GPAs for a series of students. Suppose, for example, that a student has a 3-unit 3.0, a 4-unit 2.9, a 3-unit 3.2 and a 2-unit 2.5. We can compute an overall GPA that is weighted by the individual units for each course.

So we might have an input file that has its data on pairs of lines. For each pair the name will appear on the first line and the grade data will appear on the second line. For example, we might have an input file that looks like this:

```
Erica Kane
3 2.8 4 3.9 3 3.1
Greenlee Smythe
3 3.9 3 4.0 4 3.9
Ryan Laveree
2 4.0 3 3.6 4 3.8 1 2.8
Adam Chandler
3 3.0 4 2.9 3 3.2 2 2.5
Adam Chandler, Jr
4 1.5 5 1.9
```

This data is line-based, but some of the lines have individual tokens that we'll need to process. So we'll want to use a slight variation of the template given in the last section. Recall that in general the way that we get a combination of line-based and token-based processing is to write code like this:

```
while (input.hasNextLine()) {
    String text = input.nextLine();
    Scanner data = new Scanner(text);
    <process data>
}
```

For this file, our data appears as pairs of lines where the first line stores the name and the second line stores the grade data. We won't need to process the name token by token, but we'll need to process the grades as individual tokens. So we can use the following slight variation that reads two lines of data each time through the loop and that constructs a `Scanner` object for the second line of input that contains the grade information:

```
while (input.hasNextLine()) {
    String name = input.nextLine();
    String grades = input.nextLine();
    Scanner data = new Scanner(grades);
    <process this student's data>
}
```

To complete this program, we have to figure out how to process the grade data in our `Scanner` called `data`. This is a good place to introduce a static method. The code above involves processing the overall file. The task of processing one list of grades should be split off into its own method. Let's call it processGrades. Obviously it can't do its work without the `Scanner` object that has the

grades, so we'll pass that as a parameter. What exactly needs to be done? The plan was to compute a weighted GPA for each student. So this method needs to read the individual grades and turn that into a single GPA score.

Weighted GPAs involve computing a value known as the "quality points" for each grade. The quality points are defined as the units times the grade. The weighted GPA is calculated by dividing the total quality points by the total units. So we just need to add up the total quality points and add up the total units, then divide. This involves a pair of cumulative sum tasks that we can express in pseudocode as follows:

```
set total units to 0.
set total quality points to 0.
while (more grades to process) {
    read next units and next grade.
    add next units to total units.
    add (next units) * (next grade) to total quality points.
}
set gpa to (total quality points)/(total units).
```

This is fairly simple to translate into Java code by incorporating our `Scanner` object called `data`:

```
double totalQualityPoints = 0.0;
double totalUnits = 0;
while (data.hasNextInt()) {
    int units = data.nextInt();
    double grade = data.nextDouble();
    totalUnits += units;
    totalQualityPoints += units * grade;
}
double gpa = totalQualityPoints/totalUnits;
```

Because our `Scanner` object data was constructed from a single line of input, we can process just one person's grades with this loop.

There is still a potential problem. What if there are no grades? Some students might have dropped all of their classes, for example. There are several ways we might handle that situation, but let's assume that it is appropriate to use a GPA of 0.0 when there are no grades.

Making that correction and putting this into a method, we end up with the following code.

```
public static double processGrades(Scanner data) {
    double totalQualityPoints = 0.0;
    double totalUnits = 0;
    while (data.hasNextInt()) {
        int units = data.nextInt();
        double grade = data.nextDouble();
        totalUnits += units;
        totalQualityPoints += units * grade;
    }
    if (totalUnits == 0) {
        return 0.0;
    } else {
        return totalQualityPoints/totalUnits;
    }
}
```

Recall that our high-level code looked like this:

```
while (input.hasNextLine()) {
    String name = input.nextLine();
    String grades = input.nextLine();
    Scanner data = new Scanner(grades);
    <process this student's data>
}
```

We can now start to fill in the details of what it means to "process this student's data." We will call the method we just wrote to process the grades for this student and to turn it into a weighted GPA and then print the results:

```
double gpa = processGrades(data);
System.out.println("GPA for " + name + " = " + gpa);
```

This would complete the program, but let's add one more calculation. Let's compute the max and min GPA that we see among these students. We can accomplish this fairly easily with some simple if statements after the `println`:

```
if (gpa > max) {
    max = gpa;
}
if (gpa < min) {
    min = gpa;
}
```

We simply compare the current gpa against what we currently consider the max and min, resetting if the new gpa represents a new max or a new min. But how do we initialize these variables? We have two approaches to choose from. One approach involves initializing the max and the min to the first value in the sequence. We could do that, but it would make our loop much more complicated than it is currently. The second approach involves setting the max to the lowest possible value and setting the min to the highest possible value. This approach isn't always possible because we don't always know how high or low our values might go. But in the case of GPAs, we know that they will always be between 0.0 and 4.0.

Thus, we can initialize the variables as follows:

```
double max = 0.0;
double min = 4.0;
```

It may seem odd to set the max to 0 and the min to 4, but that's because we are intending to have them reset inside the loop. If the first student has a GPA of 3.2, for example, then this will constitute a new max (higher than 0.0) and a new min (lower than 4.0). Of course, it's possible that all students end up with a 4.0, but then our choice of 4.0 for the min is appropriate. Or all students could end up with a 0.0, in which case our choice of a max of 0.0 is appropriate.

Putting this all together we get the following complete program.

```
 1  // This program reads an input file with GPA data for a series of students
 2  // and reports a weighted GPA for each.  The input file should consist of
 3  // a series of line pairs where the first line has a student's name and the
 4  // second line has a series of grade entries.  The grade entries should be
 5  // a number of units (an integer) followed by a grade (a number between 0.0
 6  // and 4.0).  For example, the input might look like this:
 7  //
 8  //     Erica Kane
 9  //     3 2.8 4 3.9 3 3.1
10  //     Ryan Laveree
11  //     2 4.0 3 3.6 4 3.8 1 2.8
12  //
13  // The program reports the weighted GPA for each student along with the
14  // max and min GPA.
15
16  import java.io.*;
17  import java.util.*;
18
19  public class Gpa {
20      public static void main(String[] args) throws FileNotFoundException {
21          Scanner input = new Scanner(new File("gpa.dat"));
22          process(input);
23      }
24
25      public static void process(Scanner input) {
26          double max = 0.0;
27          double min = 4.0;
28          while (input.hasNextLine()) {
29              String name = input.nextLine();
30              String grades = input.nextLine();
31              Scanner data = new Scanner(grades);
32              double gpa = processGrades(data);
33              System.out.println("GPA for " + name + " = " + gpa);
34              if (gpa > max) {
35                  max = gpa;
36              }
37              if (gpa < min) {
38                  min = gpa;
39              }
40          }
41          System.out.println();
42          System.out.println("max GPA = " + max);
43          System.out.println("min GPA = " + min);
44      }
45
46      public static double processGrades(Scanner data) {
47          double totalQualityPoints = 0.0;
```

```
48          double totalUnits = 0;
49          while (data.hasNextInt()) {
50              int units = data.nextInt();
51              double grade = data.nextDouble();
52              totalUnits += units;
53              totalQualityPoints += units * grade;
54          }
55          if (totalUnits == 0) {
56              return 0.0;
57          } else {
58              return totalQualityPoints / totalUnits;
59          }
60      }
61  }
```

Once again our `main` method has the `throws FileNotFoundException` in its header. This program executes as follows assuming the data above is placed in a file called `gpa.dat`.

```
GPA for Erica Kane = 3.3299999999999996
GPA for Greenlee Smythe = 3.9299999999999997
GPA for Ryan Laveree = 3.6799999999999997
GPA for Adam Chandler = 2.9333333333333336
GPA for Adam Chandler, Jr = 1.7222222222222223

max GPA = 3.9299999999999997
min GPA = 1.7222222222222223
```

The program could be modified to send its output to an external file such as gpa.out. This is left as an exercise.

Chapter Summary

- A `Scanner` object can read input from a file rather than from the keyboard. This is achieved by passing a `new File(`*`filename`*`)` to the `Scanner`'s constructor, rather than passing `System.in`.
- A checked exception is a program error condition that must be checked for the program to compile. When constructing a `Scanner` that reads a file, we must write the phrase `throws FileNotFoundException` on the `main` method's header.
- Input files are treated by the `Scanner` as a one-dimensional stream of data that is read in order from start to end. The input cursor consumes (moves past) input tokens as they are read and returns them to your program.
- A file name can be specified as a relative path, such as `"data/text/numbers.dat"`, which would be assumed to exist in the `data/text/` subfolder of the current directory. Also a full file path can be specified, such as `"C:/Documents and Settings/user/My Documents/data/text/numbers.dat"`.
- `Scanner`s that read files make use of the various `hasNext` methods to discover when the file's input has been exhausted.

- Many files have their input structured by lines, and it makes sense to process those files line-by-line. In such cases, it is common to use nested loops: an outer loop over each line of the file, and an inner loop that processes the tokens in each line.
- Output to a file can be achieved with a `PrintStream` object, which is constructed with a `File` and has the same methods as System.out, such as `println` and `print`.
- Java has a `try/catch` statement that can be used to check for errors and handle them while the program is executing. A `try/catch` statement can be used when opening a file, to handle the possibility that the file does not exist.

Self-Check Problems

Section 6.1: File Reading Basics

1. What is an external file? How can we read data from an external file in Java?
2. What is wrong with the following line of code?

```
Scanner input = new Scanner("test.dat");
```

3. Write code to construct a Scanner object to read the file input.txt which exists in the same folder as your program.

Section 6.2: Details of Token-Based Processing

4. What is wrong with the following line of code?

```
Scanner input = new Scanner(new File("C:\temp\new files\test.dat"));
```

   (Hint: Try printing the above `String`.)

5. If your Java program is located on a Windows machine in the folder `C:\Documents and Settings\amanda\My Documents\programs`, answer the following:
   - What are two legal ways you can refer to the file: `C:\Documents and Settings\amanda\My Documents\numbers.dat` ?
   - What about the file: `C:\Documents and Settings\amanda\My Documents\programs\data\homework6\input.dat` ?
   - How many legal ways can you refer to the file: `C:\Documents and Settings\amanda\My Documents\homework\data.txt` ? What is/are they?
6. If your Java program is located on a Linux machine in the folder `/home/amanda/Documents/hw6`, answer the following:
   - What are two legal ways you can refer to the file `/home/amanda/Documents/hw6/names.txt` ?
   - What about the file `/home/amanda/Documents/hw6/data/numbers.txt` ?

- How many legal ways can you refer to the file `/home/amanda/download/saved.html` ? What is/are they?

Section 6.3: Line-based Processing

7. For the next several questions consider a file named `readme.txt` that contains the following contents:

```
6.7        This file has
        several input lines.

  10 20         30   40

test
```

   What would be the output from the following code when run on the above `readme.txt` file?

```
Scanner input = new Scanner(new File("readme.txt"));
int count = 0;
while (input.hasNextLine()) {
    System.out.println("input: " + input.nextLine());
    count++;
}
System.out.println(count + " total");
```

8. What would be the output from the code in the previous exercise if the calls to `hasNextLine` and `nextLine` are replaced by calls to `hasNext` and `next`, respectively?

9. What would be the output from the code in the previous exercise if the calls to `hasNextLine` and `nextLine` are replaced by calls to `hasNextInt` and `nextInt`, respectively? How about `hasNextDouble` and `nextDouble`?

10. Write a program that prints itself to the console as output. That is, if the program is stored in Example.java, it will open the file Example.java and print its contents.

11. Write code that prompts the user for a file name, and prints the contents of that file to the console as output. Assume that the file exists. You may wish to place this code into a method named `printEntireFile`.

Section 6.4: Advanced File Processing

12. What object is used to write output into an external file? What methods does this object have available for you to use?

13. Write code to print the following four lines of text into a file named message.txt:

```
Testing,
1, 2, 3.

This is my output file.
```

14. Write code that repeatedly prompts the user for a file name until the user types the name of a file that exists on the system. You may wish to place this code into a method named `getFileName`, which would return that file name as a `String`.

15. A previous problem had you write a piece of code to prompt the user for a file name and print that file's contents to the console. Modify your code so that it will repeatedly prompt the user for the file name until the user types a valid file that exists.

Exercises

1. Write a method named `doubleSpace` that accepts two `String`s representing file names as its parameters, writing into the second file a double-spaced version of the text in the first file. You can achieve this by inserting a blank line between each line of output.

   To make it more challenging, try to make your code so that it would work even if the two `String`s were the same (in other words, to write the double-spaced output back into the same file).

2. Write a method named `wordWrap` that accepts a `Scanner` representing an input file as its parameter and outputs each line of the file to the console, word-wrapping all lines that are longer than 60 characters. For example, if a line contains 112 characters, replace it by two lines: one containing the first 60 characters and another containing the final 52 characters. A line containing 217 characters would be wrapped into four lines: three of length 60 and a final line of length 37.

3. Modify the preceding word-wrap method so that it outputs the newly-wrapped text back into the original file. (Be careful -- don't output into a file while you are reading it!) Also, modify it to use a class constant for the maximum line length rather than hard-coding 60.

4. Modify the preceding word-wrap method so that it only wraps whole words, never chopping a word in half. Assume that a word is any whitespace-separated token and that all words are under 60 characters in length.

5. Write a program that prompts the user for a file name, then reads that file (assuming that its contents consist entirely of integers) and prints the maximum, minimum, sum, count (number of integers in the file), and average of the numbers. For example, if the file `numberinput.dat` has the following contents:

   ```
   4 -2 18
   15 31

   27
   ```

   Your program would produce the following output:

   ```
   What is the name of the input file? numberinput.dat
   Maximum = 31
   Minimum = -2
   Sum = 93
   Count = 6
   Average = 15.5
   ```

6. Modify the previous program so that it works even on an input file that contains non-integer tokens. Your code should skip over any tokens that are not valid integers. For example, if the file `numberinput2.dat` has the following contents:

```
4   billy bob   -2 18  2.54
15 31 NotANumber

true  'c' 27
```

   Your program would produce the same output as in the previous exercise.

7. Write a method named `collapseSpaces` that accepts a `Scanner` representing an input file as its parameter, then reads that file and outputs it with all its tokens separated by a single space. Your code will collapse sequences of multiple spaces into a single space. That is, if a line of the file contains the following text:

```
   many   spaces   on  this      line!
```

   The same line of the file would be output as follows:

```
many spaces on this line!
```

8. Write a method named `readEntireFile` that accepts a `Scanner` representing an input file as its parameter, then reads that file and returns the entire text contents of that file as a `String`.

9. Write a method named `stripHtmlTags` that accepts a `Scanner` representing an input file as its parameter, then reads that file, assuming that the file contains an HTML web page, and prints the file's text with all HTML tags removed. A tag is any text between < and > characters. For example, if the file contains the following text:

```
<html>
<head>
<title>My web page</title>
</head>
<body>
<p>There are many pictures of my cat here,
as well as my <b>very cool</b> blog page,
which contains <font color="red">awesome</font>
stuff about my trip to Vegas.<p>

Here's my cat now: <img src="cat.jpg">
</body>
</html>
```

   Your program should output the following text:

```
My web page


There are many pictures of my cat here,
as well as my very cool blog page,
which contains awesome
stuff about my trip to Vegas.

Here's my cat now:
```

You may assume that the file is a well-formed HTML document and that no tag contains a < or > character inside itself.

10. Write a method named `stripComments` that accepts a `Scanner` representing an input file as its parameter, then reads that file, assuming that the file contains a Java program, and prints the file's text with all comments removed. A tag is any text on a line from // to the end of the line, and any text between /* and */ characters. For example, if the file contains the following text:

```
import java.util.*;

/* My program
by Suzy Student */
public class Program {
    public static void main(String[] args) {

        System.out.println("Hello, world!");  // prints a message
    }

    public static /* Hello there */ void foo() {  // comment here
        System.out.println("Goodbye!");
    } /* */
}
```

Your program should output the following text:

```
import java.util.*;


public class Program {
    public static void main(String[] args) {

        System.out.println("Hello, world!");
    }

    public static  void foo() {
        System.out.println("Goodbye!");
    }
}
```

Programming Projects

1. Students are often told that their term papers should have a certain number of words in them. Counting words in a long paper is a tedious task, but the computer can help. Write a program that counts the number of words in a paper assuming that consecutive words are separated either by spaces or end-of-line characters. You could then extend the program to count not just the number of words, but the number of lines and the total number of characters in the file.

2. Write a program that takes as input lines of text like:

   ```
   This is some
   text here.
   ```

   and produces as output the same text inside a box, as in:

   ```
   +------------+
   |This is some|
   |text here.  |
   +------------+
   ```

   Your program will have to assume some maximum line length (e.g., 12 above).

3. Write a program that compares two files and prints information about the differences between them. For example, if a file `data1.txt` exists with the following contents:

   ```
   This file has a great deal of
   text in it which needs to

   be processed.
   ```

   and another file `data2.txt` exists with the following contents:

   ```
   This file has a grate deal of
   text in it which needs to

   bee procesed.
   ```

   Then a dialogue of the user running your program might look like this:

   ```
   Enter a first file name: data1.txt
   Enter a second file name: data2.txt

   Differences found:
   Line 1:
   < This file has a great deal of
   > This file has a grate deal of

   Line 4:
   < be processed.
   > bee procesed.
   ```

4. Write a program that prompts the user for a file name, assuming that the file contains a Java program. Your program should read the file and print its contents properly indented. When you see a { left brace character in the file, increase your indentation level by four spaces. When

you see a } right brace character, decrease your indentation level by four spaces. You may assume that the file has only one opening or closing brace per line, that every block statement such as `if` or `for` uses braces rather than omitting them, and that every relevant occurrence of a { or } character in the file occurs at the end of a line.

If the file `BadlyIndented.java` contains the following contents:

```
 1  // This file is poorly indented!
 2     public class BadlyIndented {
 3  public static void main(String[] args) {
 4    System.out.println("Hello, world!");
 5    myMethod();
 6    }
 7
 8           public static void myMethod() {
 9           System.out.println("How ugly I am!");
10            if (1 + 1 == 3) {
11      System.out.println("the sky is falling.");
12            }
13                     else {
14                   System.out.println("Okay.");
15     }
16        }
17   }
```

Your program should produce the following output:

```
What is the name of the input file? BadlyIndented.java

// This file is poorly indented!
public class BadlyIndented {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
        myMethod();
    }

    public static void myMethod() {
        System.out.println("How ugly I am!");
        if (1 + 1 == 3) {
            System.out.println("the sky is falling.");
        }
        else {
            System.out.println("Okay.");
        }
    }
}
```

Consider using a class constant for the number of spaces to indent (4), so that it could be easily changed later.

Stuart Reges
Marty Stepp

Chapter 7
Arrays

Copyright © 2006 by Stuart Reges and Marty Stepp

- 7.1 Array Basics
  - Constructing and Accessing an Array
  - A Useful Application of Arrays
  - Random Access
  - Arrays and Methods
  - The For-Each Loop
  - Limitations of Arrays
- 7.2 Advanced Arrays
  - Shifting Values in an Array
  - Initializing Arrays
  - Arrays in the Java Class Libraries
  - Arrays of Objects
  - Command Line Arguments
- 7.3 Multidimensional Arrays (optional)
  - Rectangular Two-Dimensional Arrays
  - Jagged Arrays
- 7.4 Case Study: Hours Worked
  - The transferFrom Method
  - The sum Method
  - The addTo Method
  - The print Method
  - The Complete Program

Introduction

The sequential nature of files severely limits the number of interesting things you can easily do with them. The algorithms you have examined so far have all been sequential algorithms: algorithms that can be performed by examining each data item once in sequence. There is an entirely different class of algorithms that can be performed when you can access the data items multiple times and in an arbitrary order.

This chapter examines a new object called an *array* that provides this more flexible kind of access The discussion of array applications begins with some sequential algorithms that are enhanced by arrays.

7.1 Array Basics

An *array* is a flexible structure for storing a sequence of values all of the same type.

> **Array**
>
> A structure that holds multiple values of the same type.

The values stored in an array are called *elements*.

Constructing and Accessing an Array

Suppose that you wanted to store three different temperature readings. You could keep them in a series of different variables:

```
double temperature1;
double temperature2;
double temperature3;
```

This isn't a bad solution if you have just three temperatures, but suppose you want three thousand temperatures. Then you would want something more flexible. You can instead store the temperatures in an array.

As we will see, arrays are objects, which means that they have to be constructed. But before we can construct an array, we need a variable that will store a reference to our array object. What type should we use for that variable? The type you use will depend on the type of elements you want to have in your array. As we will see, we use square brackets in Java to indicate that we want an array. We want a sequence of values of type double, so we use the type double[]. So we can declare a variable for storing our array as follows:

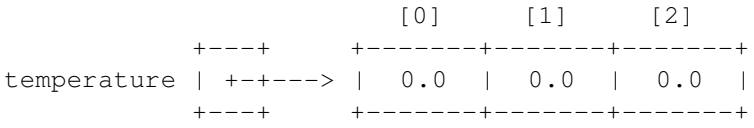
```
double[] temperature;
```

Remember that objects must be constructed. Simply declaring a variable isn't enough to bring the object into existence. In our case, we want an array of 3 double values, which we can construct as follows:

```
double[] temperature = new double[3];
```

This is a slightly different syntax than what we have seen before when we asked for a new object. It is a special syntax for arrays only. Notice that on the left-hand side we don't put anything inside the square brackets because we are describing a type. The variable called temperature can refer to any array of double values, no matter how many elements it has. On the right-hand side we have to mention a specific number of elements because we are asking Java to construct an actual array object for us and it needs to know how many elements to include.

In executing the line of code above, Java will construct an array of 3 double values with the variable temperature referring to the array. The elements are all initialized to 0.0:



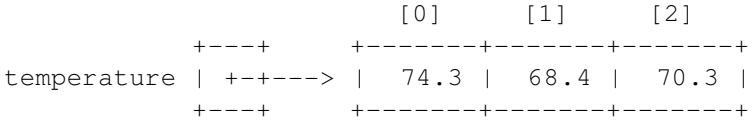
As the picture above indicates, the variable temperature is not itself the array. Instead, it stores a reference to the array.

The individual elements are indexed by integers. This is similar to the way post office boxes are set up. The boxes are indexed with numbers so that you can refer to an individual box by using a description like "PO box 884." The array indexes are indicated in the picture above inside of square brackets. So there is element known as temperature[0], an element known as temperature[1] and an element known as temperature[2]. It might seem more natural to have indexes that start with one instead of zero but Sun decided that Java would use the same zero-based indexing scheme that is used in C and C++.

Using the indexes, we can initialize each element of the array:

```
temperature[0] = 75.2;
temperature[1] = 68.4;
temperature[2] = 70.3;
```

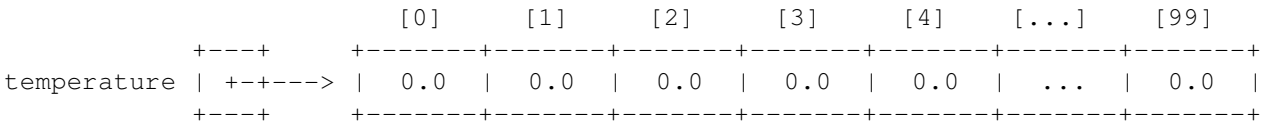
which would modify the array to have the following values:



Obviously an array isn't particularly helpful when you have just three values to store, but arrays are flexible in that you can easily request a much larger array. For example, you could request an array of one-hundred temperatures by saying:

```
double[] temperature = new double[100];
```

This is almost the same line of code we executed before. The variable is still declared to be of type double[]. But in constructing the array, we request 100 elements instead of 3, which constructs a much larger array:



Notice that the highest index is 99 rather than 100 because of zero-based indexing. You are not restricted to simple literal values inside of the brackets. You can use any integer expression. This allows you to combine arrays with loops, which greatly simplifies the code you write. For example, suppose we want to read a series of temperatures from a Scanner. We could read each value individually, as in:

```
temperature[0] = input.nextDouble();
temperature[1] = input.nextDouble();
temperature[2] = input.nextDouble();
...
temperature[99] = input.nextDouble();
```

The only thing that changes from one statement to the next is the index. We can capture this pattern in a for loop with a control variable that takes on the values 0 to 99:

```
for (int i = 0; i < 100; i++) {
    temperature[i] = input.nextDouble();
}
```

This is a very concise way to initialize all elements of the array. The code works when the array has a length of 100, but you can imagine the array having a different length. Java provides a useful mechanism for making this code more general. Each array keeps track of its own length. We are using the variable temperature to refer to our array, which means that we can ask for temperature.length to find out the length of the array. By using temperature.length in the for loop test instead of the specific value 100, we make our code more general:

```
for (int i = 0; i < temperature.length; i++) {
    temperature[i] = input.nextDouble();
}
```

This code provides a template that you will see often with array processing code: a for loop that starts at 0 and that continues while the loop variable is less than the length of the array, doing something with element [i] in the body of the loop.

It is possible to refer to an illegal index of an array, in which case Java throws an exception. For example, if we were to ask for temperature[-1] or temperature[100], we would be asking for an array element that does not exist. If your code makes such an illegal reference, Java will halt your program with an ArrayIndexOutOfBoundsException.

A Useful Application of Arrays

Let's look at a program where an array allows us to solve a problem we couldn't solve before. If you tune in to any local news broadcast at night, you'll hear them report the high temperature for that day. It is usually reported as an integer, as in, "It got up to 78 today."

Suppose that we want to examine a series of high temperatures and compute the average temperature and count how many days were above average in temperature. We have been using a Scanner to solve problems like this and we can almost solve the problem that way. If we just wanted to know the average, we could use a Scanner and write a cumulative sum loop to find the average:

```
1  // Reads a series of high temperatures and reports the average.
2
3  import java.util.*;
4
5  public class Temperature1 {
6      public static void main(String[] args) {
7          Scanner console = new Scanner(System.in);
8          System.out.print("How many days' temperatures? ");
9          int numDays = console.nextInt();
10         int sum = 0;
11         for (int i = 1; i <= numDays; i++) {
12             System.out.print("Day " + i + "'s high temp: ");
13             int next = console.nextInt();
14             sum += next;
15         }
16         double average = (double) sum / numDays;
17         System.out.println();
18         System.out.println("Average = " + average);
19     }
20 }
```

This program does a pretty good job. Here is a sample execution:

```
How many days' temperatures? 5
Day 1's high temp: 78
Day 2's high temp: 81
Day 3's high temp: 75
Day 4's high temp: 79
Day 5's high temp: 71

Average = 76.8
```

But how do we count how many days were above average? We could try to incorporate it into our loop, but that won't work. The problem is that we can't figure out the average until we have gone through all of the data. That means that we need to make a second pass through the data to figure out how many days were above average. We can't do that with a Scanner. A Scanner has no "reset" option that allows us to see the data a second time.

Fortunately, we can solve the problem with an array. As we read numbers in and compute the cumulative sum, we can fill up an array that stores the temperatures. Then we can use the array to make our second pass through the data.

In the last section we were using an array of double values, but here we want an array of int values. So instead of declaring a variable of type double[], we declare a variable of type int[]. We are asking the user how many days of temperature data to include, so we can construct the array right after we have read that information:

```
int numDays = console.nextInt();
int[] temps = new int[numDays];
```

The old loop looks like this:

```
for (int i = 1; i <= numDays; i++) {
    System.out.print("Day " + i + "'s high temp: ");
    int next = console.nextInt();
    sum += next;
}
```

Because we are using an array, we will change this to a loop that starts at 0 to match the array indexing. But just because we are using zero-based indexing inside the program, that doesn't mean that we have to confuse the user by asking for "Day 0's high temp". So we can modify the println to prompt for day (i + 1). We no longer need the variable next because we will be storing the values in the array instead. So the loop code becomes:

```
for (int i = 0; i < numDays; i++) {
    System.out.print("Day " + (i + 1) + "'s high temp: ");
    temps[i] = console.nextInt();
    sum += temps[i];
}
```

Notice that we now test whether the index is strictly less than numDays. After this loop executes, we compute the average just as we did before. Then we can write a new loop that counts how many days were above average:

```
int above = 0;
for (int i = 0; i < temps.length; i++) {
    if (temps[i] > average) {
        above++;
    }
}
```

Notice that in this loop the test involves temps.length. We could instead have tested whether the variable is less than numDays. Either choice works in this program because they should be equal to each other.

If we put these various code fragments together and include code to report the number of days above average, we get the following complete program:

```
 1  // Reads a series of high temperatures and reports the average
 2  // and the number of days above average.
 3
 4  import java.util.*;
 5
 6  public class Temperature2 {
 7      public static void main(String[] args) {
 8          Scanner console = new Scanner(System.in);
 9          System.out.print("How many days' temperatures? ");
10          int numDays = console.nextInt();
11          int[] temps = new int[numDays];
12          int sum = 0;
13          for (int i = 0; i < numDays; i++) {
14              System.out.print("Day " + (i + 1) + "'s high temp: ");
15              temps[i] = console.nextInt();
16              sum += temps[i];
17          }
18          double average = (double) sum / numDays;
19          int above = 0;
20          for (int i = 0; i < temps.length; i++) {
21              if (temps[i] > average) {
22                  above++;
23              }
24          }
25          System.out.println();
26          System.out.println("Average = " + average);
27          System.out.println(above + " days above average");
28      }
29  }
```

Below is a sample execution:

```
How many days' temperatures? 9
Day 1's high temp: 75
Day 2's high temp: 78
Day 3's high temp: 85
Day 4's high temp: 71
Day 5's high temp: 69
Day 6's high temp: 82
Day 7's high temp: 74
Day 8's high temp: 80
Day 9's high temp: 87

Average = 77.88888888888889
5 days above average
```

Common Programming Error: Off By One Bug

In converting the first version of the Temperature program to one that uses an array, we modified our for loop to start with an index of 0 instead of 1. The original for loop was written this way:

```
for (int i = 1; i <= numDays; i++) {
    System.out.print("Day " + i + "'s high temp: ");
    int next = console.nextInt();
    sum += next;
}
```

We decided that we would read the values into the array rather than reading them into a variable called next, which meant that we replaced next with temps[i].

```
// wrong loop bounds
for (int i = 1; i <= numDays; i++) {
    System.out.print("Day " + i + "'s high temp: ");
    temp[i] = console.nextInt();
    sum += temp[i];
}
```

Because the array is indexed starting at 0, we changed the bounds of the for loop to start at 0. Suppose that was the only change we made, changing 1 to 0:

```
// still wrong loop bounds
for (int i = 0; i <= numDays; i++) {
    System.out.print("Day " + (i + 1) + "'s high temp: ");
    temps[i] = console.nextInt();
    sum += temps[i];
}
```

This loop generates an error we run it. It asks for an extra day's worth of data and then throws an exception, as in the following sample execution:

```
How many days' temperatures? 5
Day 1's high temp: 82
Day 2's high temp: 80
Day 3's high temp: 79
Day 4's high temp: 71
Day 5's high temp: 75
Day 6's high temp: 83
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Temperature2.main(test.java:15)
```

The problem is that if we are going to start the for loop variable at 0, then we want to test for it being strictly less than the number of iterations we want. We changed the 1 to a 0 but left the <= test. As a result, the loop is performing an extra iteration and trying to make a reference to an array element temps[5] that doesn't exist.

This is a classic off by one error. The fix is to change the loop bounds to use a strictly less-than test:

```
// correct bounds
for (int i = 0; i < numDays; i++) {
    System.out.print("Day " + (i + 1) + "'s high temp: ");
    temps[i] = console.nextInt();
    sum += temps[i];
}
```

Random Access

Most of the algorithms we have seen have involved *sequential access*.

Sequential Access

Manipulating values in a sequential manner from first to last.

A Scanner object is often all you need for a sequential algorithm because it allows you to access data in a forwards manner from first to last. But as we have seen, there is no way to reset a Scanner back to the beginning. We just examined a sample program that uses an array to allow a second pass through the data, but even this is fundamentally a sequential approach because it involves two passes through the data.

An array is a powerful data structure that allows a more sophisticated kind of access known as *random access*:

Random Access

Manipulating values in any order whatsoever with quick access to each value.

Arrays can provide random access because they are always allocated as a contiguous block of memory. That means that the computer can quickly compute exactly where a particular value will be stored because it knows how much space each element takes up in memory and it knows that they are all allocated right next to each other in the array.

Let's explore a problem where random access is important. Suppose that a teacher gives quizes that are scored on a scale of 0 to 4 and the teacher wants to know the distribution of quiz scores. In other words, the teacher wants to know how many scores of 0 there are and how many scores of 1, how many scores of 2, how many scores of 3 and how many scores of 4. Suppose that the teacher has included all of the scores in a data file like the following:

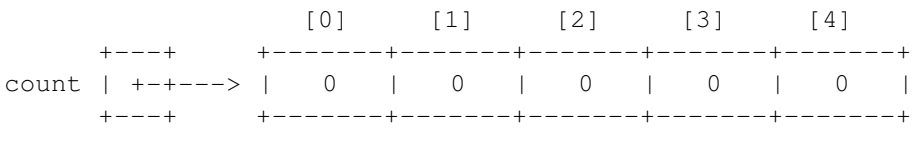
```
2 8 0 7 4 12 0 2 1 8 7 1 7 4 4 6 79 5 8 5 1 2 7 4 3 2 4
2 7 0 4 5 4 10 2 0 275 93 1 5 4 5 6 8 1 2 6 -999 9 5 7
0 64 3 4 2 3 10 2 4 9 0 5 3 -36 9 1 0 6 2 2 1 6 6 1 3 8 6
```

The teacher could hand count the scores, but it seems much easier to use a computer to do the counting. How do we solve the problem? First you have to recognize that you are doing five separate counting tasks. You are counting the occurrences of the number 0, the number 1, the number 2, the number 3, and the number 4. We will need five counters to solve this problem, which means that an array is a great way to store the data. In general, whenever you find yourself thinking that you need *n* of some kind of data, you should think about using an array of length *n*.

Each counter will be an int, so we want an array of five int values:

```
int[] count = new int[5];
```

This will allocate the array of five ints and will initialize each to 0:



We are reading from a file, so we will need a Scanner and a loop that reads scores until there are no more scores to read:

```
Scanner input = new Scanner(new File("tally.dat"));
while (input.hasNextInt()) {
    int next = input.nextInt();
    // process next
}
```

To complete this code, we need to figure out how to process each value. We know that next will be one of 5 different values: 0, 1, 2, 3 or 4. If it is 0, we want to increment our counter for 0, which is count[0]. If it is 1, we want to increment our count for 1, which is count[1]. And so on. We have been solving problems like this one with nested if/else statements:

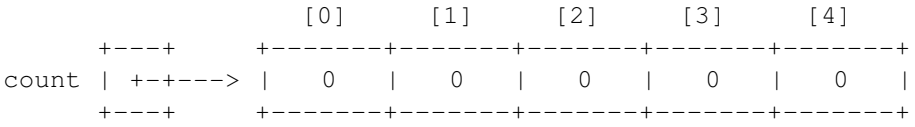
```
if (next == 0) {
    count[0]++;
} else if (next == 1) {
    count[1]++;
} else if (next == 2) {
    count[2]++;
} else if (next == 3) {
    count[3]++;
} else {  // next == 4
    count[4]++;
}
```

With an array, we can solve this much more directly:

```
count[next]++;
```

This line of code is so short compared to the nested if/else that you might not realize at first that it does the same thing. Let's simulate exactly what happens as we read various values from the file.

When the array is constructed, all of the counts are initialized to 0:



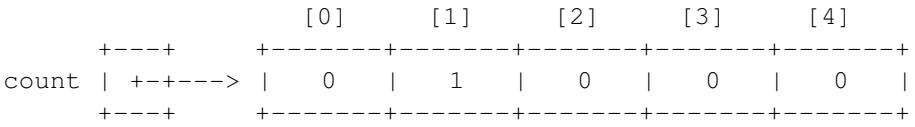
The first value in the input file is a 1, so we read that into next. Then we execute this line of code:

```
count[next]++;
```

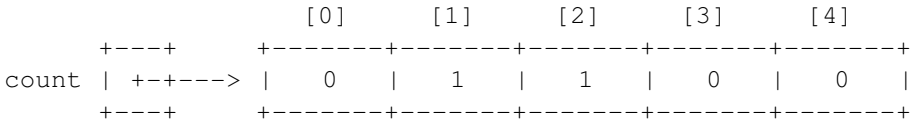
Because next is 1, this becomes:

```
count[1]++;
```

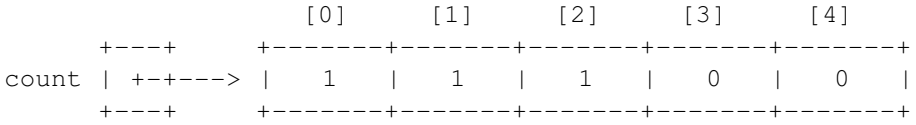
So we increment the counter at index [1]:



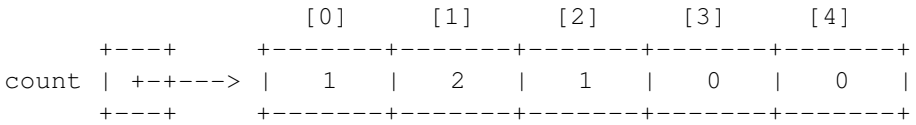
Then we read a 2 from the input file, which means we increment count[2]:



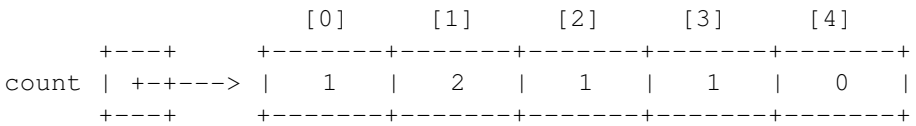
Then we read a 0 from the input file, which increments count[0]:



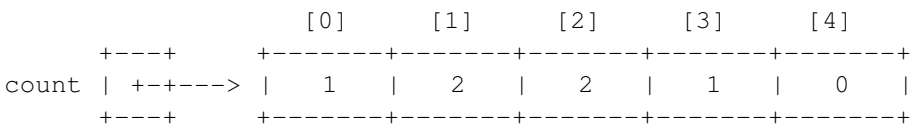
Then we read another 1 from the input file, which increments count[1]:



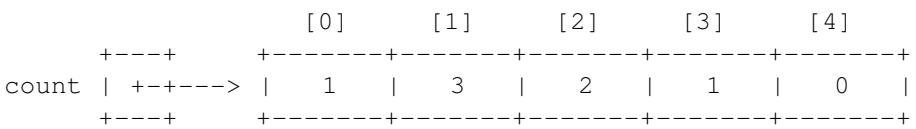
Then we read a 3 from the input file, which increments count[3]:



Then we read another 2 from the input file, which increments count[2]:



Then we read another 1 from the input file, which increments count[1]:



The program continues executing in this manner, incrementing the appropriate counter for each score it reads. Notice that the scores can appear in any order whatsoever. We can be incrementing the first value in the array one moment and incrementing the last value a moment later. Then we can jump to the middle and back to the front and back to the end and back to the middle. Any order whatsoever will work. This ability to jump around in the data structure is what we mean by random access.

After this loop finishes executing, we can report the total for each score.

```
for (int i = 0; i < count.length; i++) {
    System.out.println(i + "\t" + count[i]);
}
```

We can put this all together and add a header for the output to get the following complete program.

```
 1  // Reads a series of values and reports the frequency of occurrence
 2  // of each value.
 3
 4  import java.io.*;
 5  import java.util.*;
 6
 7  public class Tally {
 8      public static void main(String[] args) throws FileNotFoundException {
 9          Scanner input = new Scanner(new File("tally.dat"));
10          int[] count = new int[5];
11          while (input.hasNextInt()) {
12              int next = input.nextInt();
13              count[next]++;
14          }
15          System.out.println("Value\tOccurrences");
16          for (int i = 0; i < count.length; i++) {
17              System.out.println(i + "\t" + count[i]);
18          }
19      }
20  }
```

Given our sample input file, it produces the following output.

```
Value        Occurrences
0        5
1        9
2        7
3        9
4        10
```

It is important to realize that the program written with an array is much more flexible than the kind of programs we have written with simple variables and if/else statements. For example, suppose that we want to adapt this program to process an input file with exam scores that range from 0 to 100. The only change we would would have to make would be to allocate a larger array:

```
int[] count = new int[101];
```

If we had written the program with an if/else approach, we would have to add 96 new branches to account for the new range of values. With the array solution, we just have to modify the overall size of the array. Notice that the array size is one more than the highest score (101 rather than 100) because the array is zero-based and because there actually are 101 different scores that you can get on the test when 0 is a possibility.

Arrays and Methods

Arrays can be passed as parameters to methods and can be returned by methods. You have to keep in mind how to describe the array type. We have seen an array of double values that we declared to be of type double[] and an array of int values that we declared to be of type int[]. In general, for any type in Java, you can put square brackets after the type to indicate an array of that type. An array of char values is of type char[], an array of boolean is of type boolean[], an array of String values is of type String[], an array of Point objects is of type Point[] and so on.

We can explore the use of arrays as parameters by rewriting the Tally program to have methods. The program begins by constructing a Scanner and an array and then it has two loops: one to read the input file and one to report the results. We can put each loop in its own method. The first reads from the Scanner and stores its result in the array, so it will need both objects as parameters. The second reports the values in the array, so it needs just the array as a parameter. Thus, the main method would be rewritten as follows:

```
public static void main(String[] args) throws FileNotFoundException {
    Scanner input = new Scanner(new File("tally.dat"));
    int[] count = new int[5];
    readData(input, count);
    reportResults(count);
}
```

To write the readData method we just need to move the file processing loop into the method and provide an appropriate header. For the array parameter, we list the type as int[]:

```
public static void readData(Scanner input, int[] count) {
    while (input.hasNextInt()) {
        int next = input.nextInt();
        count[next]++;
    }
}
```

Many novices think that this method needs to return the array because it is changing the values stored in the array. That's not true because arrays are objects. We have seen that methods can change the state of a Scanner object, for example, simply by having the Scanner passed as a parameter. The readData method will be passed a reference to the array and that is all it needs to change the values stored inside the array.

The only time we would need to return the array would be if the method constructs the array. In that case, only the method will have a reference to the newly constructed array unless it returns it. We can rewrite the readData method to do exactly this. In its current form, it assumes that the array has already been constructed, which is why we wrote these two lines of code in main:

```
int[] count = new int[5];
readData(input, count);
```

We could instead decide that the method will construct the array. In that case, it doesn't have to be passed as a parameter, but it will have to be returned by the method. We would rewrite these two lines of code from main as a single line:

```
int[] count = readData(input);
```

And we would rewrite the method so that it constructs and returns the array:

```
public static int[] readData(Scanner input) {
    int[] count = new int[5];
    while (input.hasNextInt()) {
        int next = input.nextInt();
        count[next]++;
    }
    return count;
}
```

Pay close attention to the header of this method. It no longer has the array as a parameter and its return type is int[] rather than void. It also ends with a return statement that returns a reference to the array that it constructs.

If we combine this new version of the method with an implementation of the reportResults method, we end up with the following complete program:

```
 1  // Variation of Tally program with methods.
 2
 3  import java.io.*;
 4  import java.util.*;
 5
 6  public class Tally2 {
 7      public static void main(String[] args) throws FileNotFoundException {
 8          Scanner input = new Scanner(new File("tally.dat"));
 9          int[] count = readData(input);
10          reportResults(count);
11      }
12
13      public static int[] readData(Scanner input) {
14          int[] count = new int[5];
15          while (input.hasNextInt()) {
16              int next = input.nextInt();
17              count[next]++;
18          }
19          return count;
20      }
21
22      public static void reportResults(int[] count) {
23          System.out.println("Value\tOccurrences");
24          for (int i = 0; i < count.length; i++) {
25              System.out.println(i + "\t" + count[i]);
26          }
27      }
28  }
```

The For-Each Loop

Java 5 introduced a new loop construct that simplifies certain array loops. It is known as the enhanced for loop or the for-each loop. It can be used whenever you find yourself wanting to examine each value in the array n For example, in the program Temperature2 we had an array variable called temps and we wrote the following loop:

```
for (int i = 0; i < temps.length; i++) {
    if (temps[i] > average) {
        above++;
    }
}
```

We can rewrite this as a for-each loop:

```
for (int n : temps) {
    if (n > average) {
        above++;
    }
}
```

This loop is normally read as, "For each int n in temps...". The basic syntax of the for-each loop appears below.

```
for (<type> <name> : <array>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

There is nothing special about the variable name as long as you are consistent in the body of the loop. For example, the previous loop above could also be written with the variable x instead of the variable n:

```
for (int x : temps) {
    if (x > average) {
        above++;
    }
}
```

The for-each loop is most useful when you simply want to examine each value in sequence. There are many situations where the for-each loop is not appropriate. For example, the following loop would double every value in an array called list:

```
for (int i = 0; i < list.length; i++) {
    list[i] *= 2;
}
```

You might imagine that you could replace this with a for-each loop:

```
for (int n : list) {
    n *= 2;  // changes only n, not the array
}
```

Unfortunately, you can not modify the array elements with a for-each loop. As the comment indicates, the loop above doubles the variable n without changing the array elements.

In some cases the for-each loop is not the most convenient even when the code involves examining each array element in sequence. Consider, for example, this loop from the Tally program:

```
for (int i = 0; i < count.length; i++) {
    System.out.println(i + "\t" + count[i]);
}
```

A for-each loop can be used to replace the array access, as in:

```
for (int n : count) {
    System.out.println(i + "\t" + n);  // illegal
}
```

The problem above is that we want to print the value of i but we eliminated i when we converted this to a for-each loop. We would have to add extra code to keep track of the value of i, as in:

```
int i = 0;
for (int n : count) {
    System.out.println(i + "\t" + n);
    i++;
}
```

This code is legal, but you have to wonder whether the for-each loop is really simplifying things. In this case the original version is probably more clear.

Limitations of Arrays

You should be aware of some general limitations of arrays.

- You can't change the size of an array in the middle of program execution. To make an array bigger, you'd have to construct a new array that is larger than the old one and copy values from the old to the new array.
- You can't compare arrays for equality using a simple == test. Remember that arrays are objects, so if you ask whether one array is == to another array, you are asking whether they are the same object, not whether they store the same values. You would have to write your own comparison method that would determine whether two arrays store the same values.
- You can't print an array using a simple print or println statement. You will get an odd output when you do so. To print the contents of an array, you have to write your own for loop that prints each individual component.
- You can't change the size of an array after it has been allocated. You can't, for example, try to change the value of the length field of an array. If you find that you need a larger array, you would have to allocate a new array and copy values from the old array to the new one.

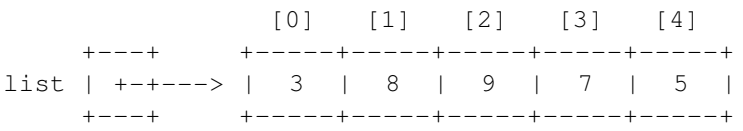
7.2 Advanced Arrays

In this section we'll discuss some advanced uses of arrays, such as a shortcut syntax for initializing an entire array with one statement. We'll also see how to create arrays that store objects instead of primitive values and we'll see how to write code that will shift values in an array.

Shifting Values in an Array

We often want to move a series of values in an array. For example, suppose we have an array of integers that stores the sequence of values (3, 8, 9, 7, 5) and we want to rotate the values so that the value at the front of the list goes to the back and the order of the other values stays the same. In other words, we want to move the 3 to the back, yielding the list (8, 9, 7, 5, 3). Let's explore how to write code to perform that action.

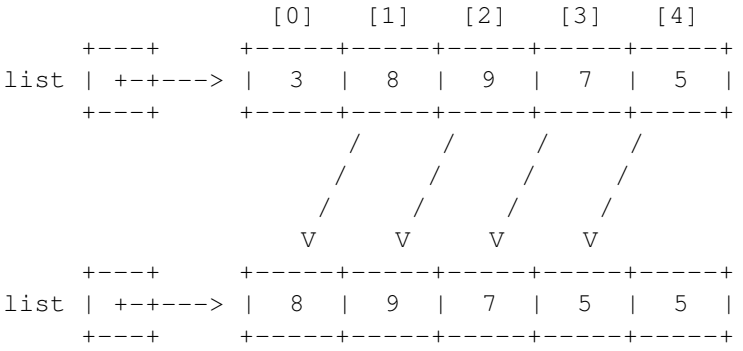
Suppose that we have a variable of type int[] called list of length 5 that stores the values (3, 8, 9, 7, 5):



The 3 at the front of the list is supposed to go to the back of the list and the other values are supposed to rotate. We can make the task easier by storing the value at the front of the list (the 3 in our example) into a local variable:

```
int first = list[0];
```

With that value safely tucked away, what we have left to do is to shift the other four values left by one position:



The overall task breaks down into four different shifting operations, each of which is a simple assignment statement:

```
list[0] = list[1];
list[1] = list[2];
list[2] = list[3];
list[3] = list[4];
```

Obviously we'd want to write this as a loop rather than writing a series of individual assignment statements. Each of the statements above is of the form:

```
list[i] = list[i + 1];
```

We replace list element [i] with the value currently stored in list element [i + 1], which shifts that value to the left. We want this inside of a for loop. Many array operations that involve accessing each element of the array can be written using a for loop that looks like this:

```
for (int i = 0; i < list.length; i++) {
    do something with list[i].
}
```

That's not a bad starting point for this operation. So we can use the assignment statement we came up with inside of this for loop body:

```
for (int i = 0; i < list.length; i++) {
    list[i] = list[i + 1];
}
```

This is almost the right answer, but it has a classic problem known as an *off by one bug*. This loop will execute 5 times for our sample array, but we want to shift just four different values. We want to do this for i equal to 0, 1, 2 and 3, but not for i equal to 4. So this loop goes one too many times. On the last iteration of the loop when i is equal to 4 we execute this line of code:

```
list[i] = list[i + 1];
```

which becomes:

```
list[4] = list[5];
```

There is no value list[5] because the array has only 5 elements with indexes 0 through 4. So this code generates an ArrayIndexOutOfBoundsException. To fix the problem, we alter the loop so that it stops one early:

```
for (int i = 0; i < list.length - 1; i++) {
    list[i] = list[i + 1];
}
```

Notice that in place of the usual list.length we use (list.length - 1). That causes the loop to stop one early. You can think of the minus one in this expression as offsetting the plus one in the assignment statement.

Of course, there is one detail left to deal with. After shifting values to the left, we have made room at the end of the list for the value that used to be at the front of the list. Recall that we stored it in a local variable called first. But we have to actually place it there after the loop executes:

```
list[list.length - 1] = first;
```

Putting all of this together into a static method we get:

```
public static void rotateLeft(int[] list) {
    int first = list[0];
    for (int i = 0; i < list.length - 1; i++) {
        list[i] = list[i + 1];
    }
    list[list.length - 1] = first;
}
```

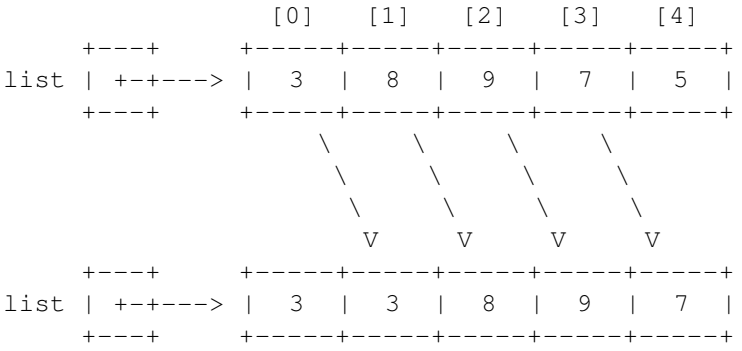
Notice that this is written as a void method. We don't need to return the array even though we have changed it. Because arrays are objects, the method is passed a reference to the location of the array. In other words, it knows where the array lives. Thus, any changes that are made by the method are changing the array object itself.

An interesting variation is to rotate the values to the right instead of rotating to the left, which is the inverse operation. So in this case, we want to take the value that is currently at the end of the list and bring it to the front. So if a variable called list stores the values (3, 8, 9, 7, 5) beforehand, then it should bring the 5 to the front and store the values (5, 3, 8, 9, 7).

We can again begin by tucking away the value that is being rotated into a temporary variable:

```
int last = list[list.length - 1];
```

Then we want to shift values to the right:



In this case the four individual assignment statements would be:

```
list[1] = list[0];
list[2] = list[1];
list[3] = list[2];
list[4] = list[3];
```

Or written more generally:

```
list[i] = list[i - 1];
```

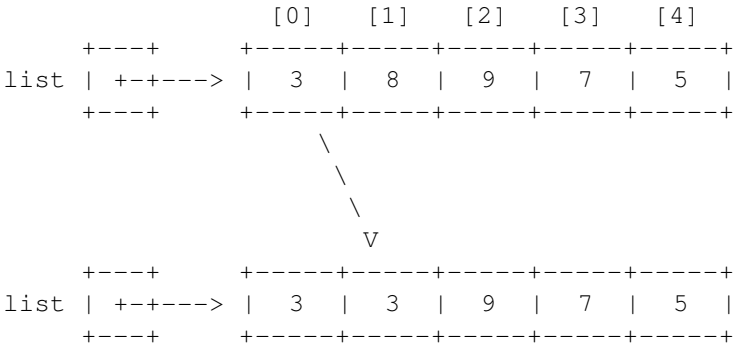
If we put this inside of the standard for loop we get:

```
for (int i = 0; i < list.length; i++) {
    list[i] = list[i - 1];
}
```

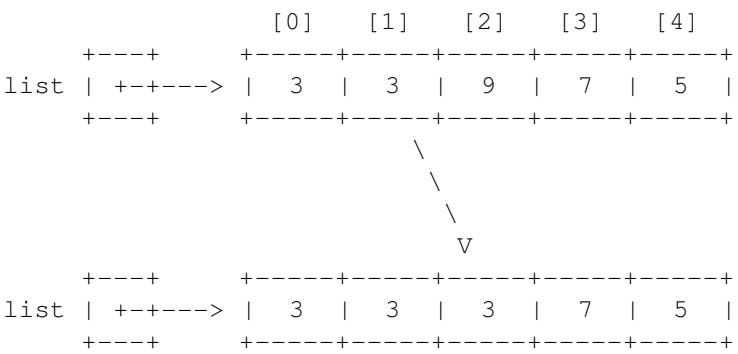
There are two problems with this code. First, there is another off-by-one bug. The first assignment statement we want to perform would set list[1] to be what is currently in list[0], but this loop sets list[0] to list[-1]. That generates an ArrayIndexOutOfBoundsException because there is no value list[-1]. So we don't want to start i at 0, we want to start it at 1:

```
for (int i = 1; i < list.length; i++) {
    list[i] = list[i - 1];
}
```

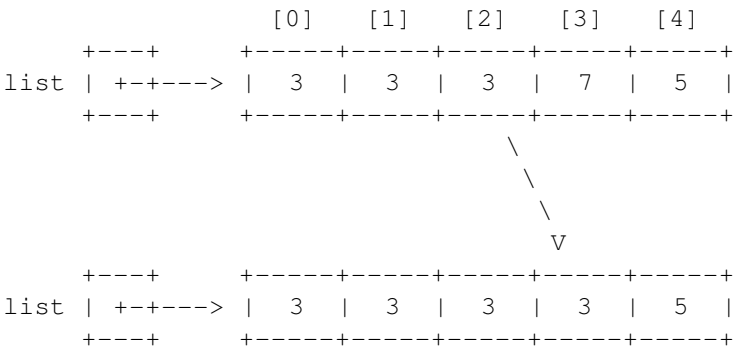
This doesn't work either. It avoids the ArrayIndexOutOfBoundsException, but think about what it does. The first time through the loop it assigns list[1] to what is in list[0]:



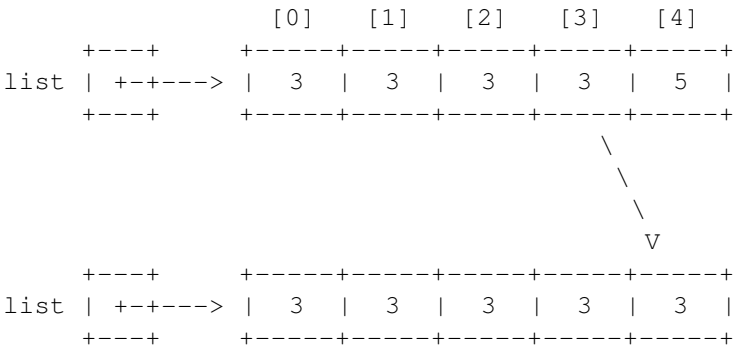
What happened to the value 8? We have overwritten it with the value 3. So the next time through the loop we set list[2] to be list[1]:



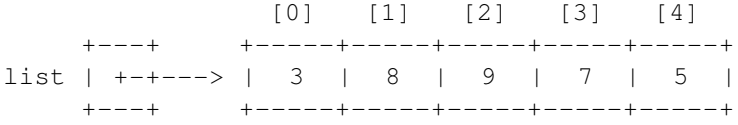
You might say, "Wait a minute...list[1] isn't a 3, it's an 8." It was an 8 when we started, but the first iteration of the loop replaced the 8 with a 3. And now we've copied that 3 into the spot where 9 used to be. So the next time through the loop we set list[3] to be what is in list[2]:



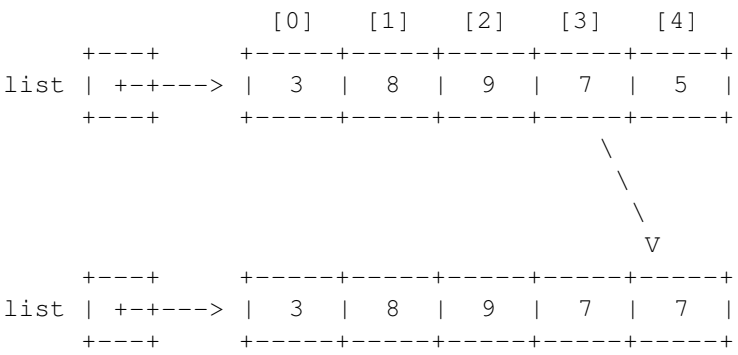
This overwrites the 7 with a 3. The final time through the loop we set list[4] to be what is currently in list[3]:



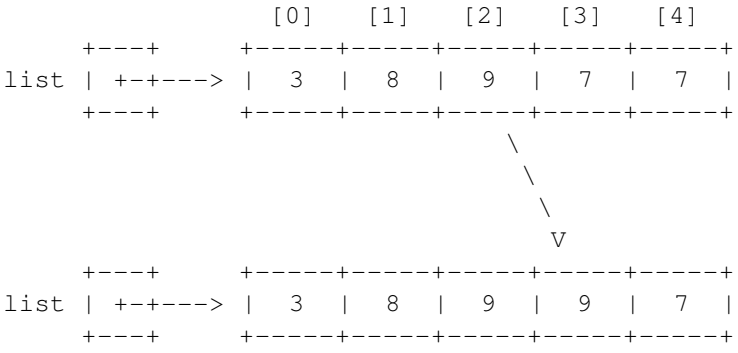
So this loop ends up putting 3 into every cell of the array. Obviously that is not what we want. To make this code work, we have to run the loop in reverse order (from right-to-left instead of left-to-right). So let's back up to where we started:



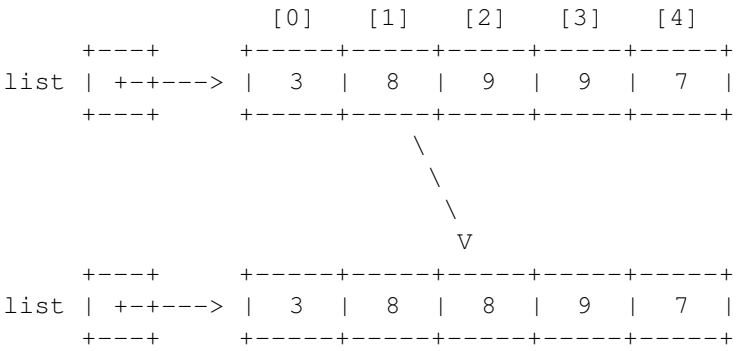
We tucked away the final value of the list into a local variable. That frees up the final array position. So we start by first assigning list[4] to be what is in list[3]:



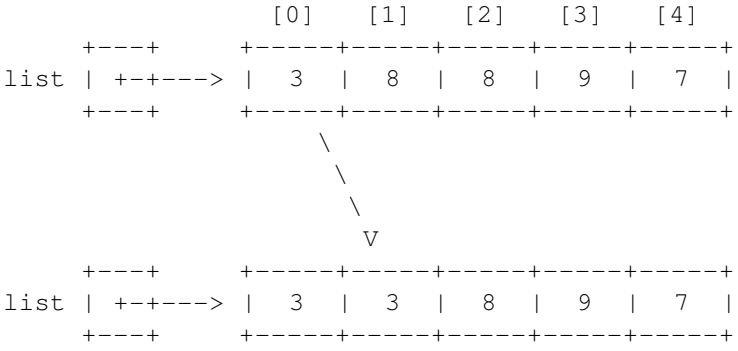
This wipes out the 5 that was a the end of the list, but we have that value tucked away in a local variable. And once we have performed this assignment statement, we free up list[3], which means now we can set list[3] to be what is currently in list[2]:



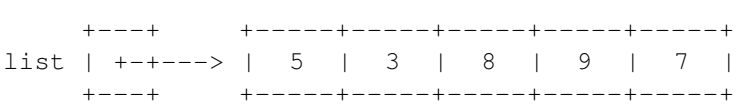
Now list[2] is free, so we can set list[2] to be what is currently in list[1]:



And we end by setting list[1] to be what is currently in list[0]:



At this point, the only thing left to do is to put the 5 that we tucked away into a local variable at the front of the list and we're done:



We can reverse the for loop by changing the i++ to i-- and adjusting the initialization and test. Putting all of this together, we get the following method:

```
public static void rotateRight(int[] list) {
    int last = list[list.length - 1];
    for (int i = list.length - 1; i > 0; i--) {
        list[i] = list[i - 1];
    }
    list[0] = last;
}
```

Initializing Arrays

Java has a special syntax for initializing an array when you know exactly what you want to put into it. For example, you could write the following code to initialize an array of ints and an array of Strings.

```
int[] numbers = new int[5];
numbers[0] = 13;
numbers[1] = 23;
numbers[2] = 480;
numbers[3] = -18;
numbers[4] = 75;

String[] words = new String[6];
words[0] = "hello";
words[1] = "how";
words[2] = "are";
words[3] = "you";
words[4] = "feeling";
words[5] = "today";
```

This works, but it's a rather tedious way to declare these arrays. Java provides a shorthand:

```
int[] numbers = {13, 23, 480, -18, 75};
String[] words = {"hello", "how", "are", "you", "feeling", "today"};
```

You use the curly braces to enclose a series of values that will be stored in the array. Order is important. The first value will go into index 0, the second value will go into index 1 and so on. Java counts how many values you include and constructs an array of just the right size. It then stores the various values into the appropriate spots in the array.

This is one of only two places where Java will construct an object without the `new` keyword. The other place we saw this was with String literals where Java constructs String objects for you without you having to call new. Both of these are conveniences for programmers because these tasks are so common that the designers of the language wanted to make it easy to do them.

Arrays in the Java Class Libraries

You will find arrays appearing throughout the Java class libraries. In fact, there is a special class called Arrays that has a collection of utility programs for manipulating arrays. One particularly handy method in the Arrays class is the method sort. For example, given the following variable declarations:

```
int[] numbers = {13, 23, 480, -18, 75};
String[] words = {"hello", "how", "are", "you", "feeling", "today"};
```

We could make the following calls:

```
Arrays.sort(numbers);
Arrays.sort(words);
```

If you then examine the values of the arrays, you will see that the numbers appear in numerically increasing order and the words appear in alphabetically increasing order. Below is a complete program that demonstrates this. The Arrays class is part of the java.util package, so you will see that the program includes an import from java.util.

```
 1  import java.util.*;
 2
 3  public class SortingExample {
 4      public static void main(String[] args) {
 5          int[] numbers = {13, 23, 480, -18, 75};
 6          String[] words = {"hello", "how", "are", "you", "feeling", "today"};
 7
 8          System.out.println("Before sorting:");
 9          display(numbers, words);
10
11          Arrays.sort(numbers);
12          Arrays.sort(words);
13
14          System.out.println("After sorting:");
15          display(numbers, words);
16      }
17
18      public static void display(int[] numbers, String[] words) {
19          for (int n : numbers) {
20              System.out.print(n + " ");
21          }
22          System.out.println();
23
24          for (String s : words) {
25              System.out.print(s + " ");
26          }
27          System.out.println();
28
29          System.out.println();
30      }
31  }
```

Notice that we were able to use for-each loops in the display method. The program produces the following output:

```
Before sorting:
13 23 480 -18 75
hello how are you feeling today

After sorting:
-18 13 23 75 480
are feeling hello how today you
```

Arrays appear in many other places in the Java class libraries. For example, the String class has a method called toCharArray that will construct an array of characters that stores the individual characters of the array. There is a corresponding constructor for the String class that allows you to construct a String from a character array. As an example, the code below converts a word into a corresponding character array, then sorts the characters in the array, then recombines the characters to make a string:

```
public static String sorted(String s) {
    char[] letters = s.toCharArray();
    Arrays.sort(letters);
    String result = new String(letters);
    return result;
}
```

This might seem like a strange method to define, but it can be very helpful for solving word puzzles known as anagrams. An anagram of a word is a word that has the same letters but in a different order. For example, the following words are all anagrams of each other: pears, spear, pares, reaps, spare. Using the method above, all of these words would be converted to `"aeprs"`. So to solve an anagram, you can simply go through a dictionary, converting every word to its sorted form and see if it matches the sorted form of the anagram you are working with.

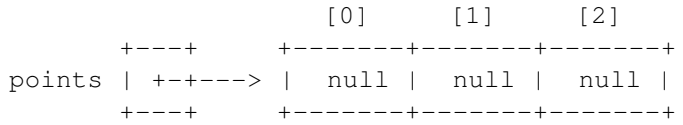
Arrays of Objects

All of the arrays we have looked at so far have stored primitive values like simple integer values, but you can have arrays of any Java type. Arrays of objects behave slightly differently because of the fact that objects are stored as references rather than as data values. Constructing an array of objects is usually a two-step process because we normally have to construct both the array and the individual objects.

Consider, for example, the following statement:

```
Point[] points = new Point[3];
```

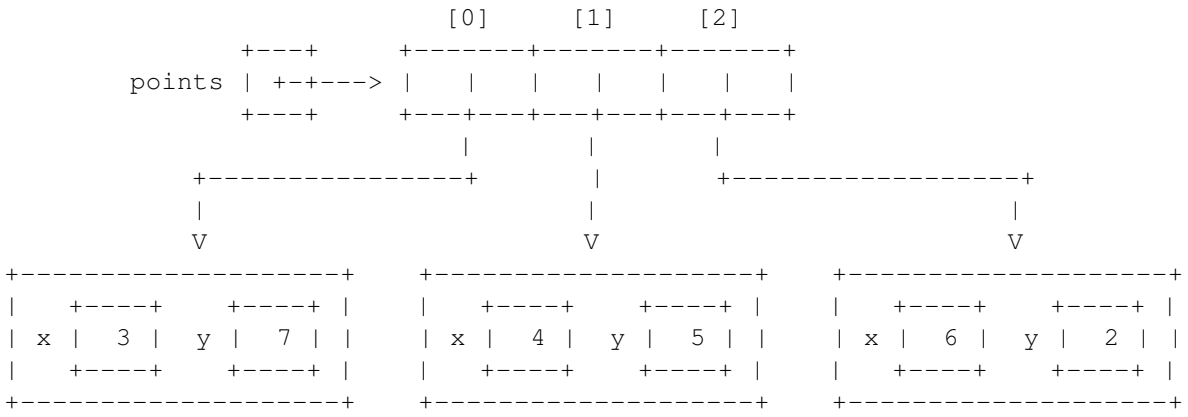
This declares a variable called points that refers to an array of length 3 that stores references to Point objects. The `new` keyword constructs the array, but it doesn't construct any actual Point objects. Instead it constructs an array of length 3 each element of which can store a reference to a Point. When Java constructs the array, it initializes these array elements to a special value known as `null` that indicates that the variable does not currently point to any object.



If we want to have some actual Point objects, they have to be constructed separately with the `new` keyword, as in:

```
Point[] points = new Point[3];
points[0] = new Point(3, 7);
points[1] = new Point(4, 5);
points[2] = new Point(6, 2);
```

After these lines of code execute, we would have individual Point objects that the various array elements refer to:



Notice that we need the `new` keyword in four different places because there are four objects to be constructed: the array itself and the three individual Point objects. We could also use the curly brace notation for initializing the array in which case we don't need the `new` keyword to construct the array itself:

```
Point[] points = {new Point(3, 7), new Point(4, 5), new Point(6, 2)};
```

Command Line Arguments

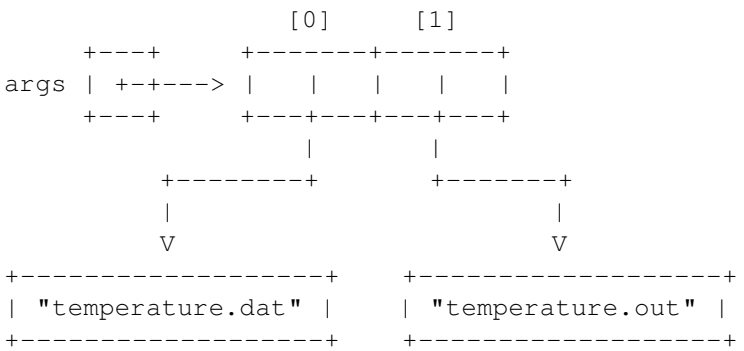
We have seen a reference an array of objects ever since we wrote the "hello world" program of chapter 1. Whenever we define a main method, we are required to include as its parameter `String[] args`, which is an array of String objects. This array is initialized by Java itself if the user provides what are known as *command line arguments* when Java is invoked. For example, normally a Java class called DoSomething would be started from the command interface by a command like this:

```
java DoSomething
```

The user has the option to type extra arguments, as in:

```
java DoSomething temperature.dat temperature.out
```

In this case the user has specified two extra arguments that are file names that the program should use (e.g., the names of an input and output file). If the user types these extra arguments when starting up Java, then the `String[] args` parameter to main will be initialized to an array of length 2 storing these two strings:



7.3 Multidimensional Arrays (optional)

The array examples in the previous sections all involved what are known as one-dimensional arrays (a single row or a single column of data). Often we want to store data in a multidimensional way. For example, we might want a two-dimensional grid of data that has both rows and columns. Fortunately, you can form arrays of arbitrarily many dimensions:

- `double`: one double
- `double[]`: a one-dimensional array of doubles
- `double[][]`: a 2-dimensional grid of doubles
- `double[][][]`: a 3-dimensional collection of doubles
- ...

Arrays of more than one dimension in general are called *multidimensional arrays*.

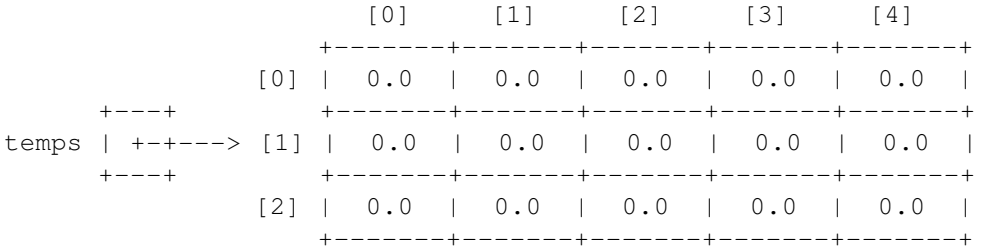
> **Multidimensional Array**
>
> An array of arrays, the elements of which are accessed with multiple integer indexes.

Rectangular Two-Dimensional Arrays

The most common use of a multidimensional array is a two-dimensional array of a certain width and height. For example, suppose that on three separate days you took a series of five temperature readings. You can define a two-dimensional array that has three rows and five columns as follows:

```
double[][] temps = new double[3][5];
```

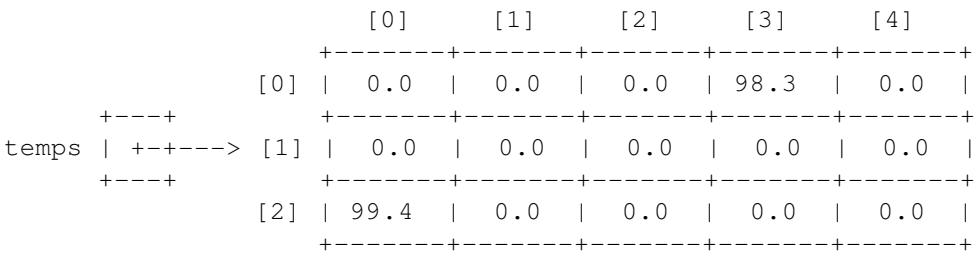
Notice that on both the left and right sides of this assignment statement we have to use a double set of square brackets. On the left in describing the type we have to make it clear that this is not just a one-dimensional sequence of values, which would be of type double[], but instead a two-dimensional grid of values which is of type double[][]. On the right in constructing the array we have to specify the dimensions of the grid. The normal convention is to list the row first followed by the column, so we ask for 3 rows and 5 columns. The resulting array would look like this:



As with one-dimensional arrays, the values are initialized to 0 and the indexes start with 0 for both rows and columns. Once we have created such an array, we can refer to individual elements by providing specific row and column numbers (in that order). For example, to set the fourth value of the first row to 98.3 and to set the first value of the third row to 99.4, we would say:

```
temps[0][3] = 98.3;  // fourth value of first row
temps[2][0] = 99.4;  // first value of third row
```

After executing these lines of code, the array would look like this:



It is helpful to think of this in a stepwise fashion starting with the name of the array. For example, if you want to refer to the first value of the third row, you obtain that through the following steps:

```
temps          the entire grid
temps[2]       the entire third row
temps[2][0]    the first element of the third row
```

You can pass multidimensional arrays as parameters just as you pass one-dimensional arrays. You need to be careful about the type. To pass the temperature grid, you would have to use a parameter of type double[][] (with both sets of brackets). For example, below is a method that prints the grid.

```
public static void print(double[][] grid) {
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            System.out.print(grid[i][j] + " ");
        }
        System.out.println();
    }
}
```

Notice that to ask for the number of rows we ask for grid.length and to ask for the number of columns we ask for grid[i].length.

Arrays can have as many dimensions as you want. For example, if you wanted a three-dimensional cube of integers that was a 4 by 4 by 4 cube, you would say:

```
int[][][] numbers = new int[4][4][4];
```

The normal convention would be to assume this is plane number followed by row number followed by column number, although you can use any convention you want as long as your code is written consistently.

Jagged Arrays

The previous examples have involved rectangular grids that have a fixed number of rows and columns. It is also possible to create a jagged array where the number of columns varies from row to row. For example, suppose that you wanted to have a data structure that stores the rows of Pascal's Triangle. Pascal's Triangle contains what are known as the binomial coefficients. The nth row of Pascal's Triangle contains the coefficients obtained when you expand:

$$(x + y)^n$$

Pascal's triangle looks like this:

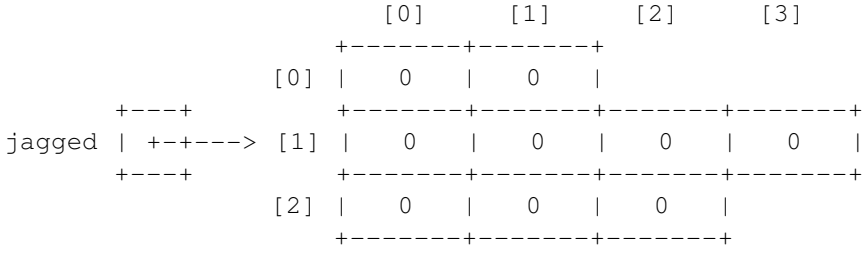
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1

The triangle has an interesting property in that each row begins and ends with a 1 and the other numbers in the row are each the sum of two numbers from the previous row. These observations can be used to compute each row in a fairly straightforward manner. But each row has a different number of elements. Row 0 has 1 column, row 1 has 2 columns, row 2 has 3 columns and so on.

To construct a jagged array we divide the construction into two steps. We construct the array for holding rows first and then we construct each individual row second. For example, to construct an array that has 2 elements in the first row, 4 elements in the second row and 3 elements in the third row, we can say:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[4];
jagged[2] = new int[3];
```

This would construct an array that looks like this:



Putting this all together, below is a program that computes the first 11 rows of Pascal's triangle. In the main method it does the first part of the construction, specifying the number of rows to use. It then calls a method fillIn that constructs each individual row. It then passes the result to a print method that prints the result.

```
1  // This program constructs a jagged two-dimensional array that stores Pascal's
2  // Triangle.  It takes advantage of the fact that each value other than the
3  // 1's that appear at the beginning and end of each row is the sum of two
4  // values from the previous row.
5
6  public class PascalsTriangle {
7       public static void main(String[] args) {
8           int[][] triangle = new int[11][];
9           fillIn(triangle);
10          print(triangle);
11       }
12
13      public static void fillIn(int[][] triangle) {
14          for (int i = 0; i < triangle.length; i++) {
15              triangle[i] = new int[i + 1];
16              triangle[i][0] = 1;
17              triangle[i][i] = 1;
18              for (int j = 1; j < i; j++) {
19                  triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
20              }
21          }
22      }
23
24      public static void print(int[][] triangle) {
25          for (int i = 0; i < triangle.length; i++) {
26              for (int j = 0; j < triangle[i].length; j++) {
27                  System.out.print(triangle[i][j] + " ");
28              }
29              System.out.println();
30          }
31      }
32 }
```

It produces the following output:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

7.4 Case Study: Hours Worked

Let's look at a more complex program example that involves using arrays. Suppose that you have an input file that has data indicating how many hours an employee has worked with each line of the input file indicating the hours worked for a different week. Each week has 7 days, so there could be up to 7 numbers listed on each line. We generally consider Monday to be the start of the work week, so let's assume that each line lists hours worked on Monday followed by hours worked

on Tuesday, and so on, and ending with hours worked on Sunday. But we'll allow the lines to have fewer than 7 numbers if someone worked fewer than 7 days. Below is a sample input file that we'll call `hours.txt`:

```
8 8 8 8 8
8 4 8 4 8 4 4
8 4 8 4 8
3 0 0 8 6 4 4
8 8
0 0 8 8
8 8 4 8 4
```

Let's write a program that reads this input file, reporting totals for each row and each column. The totals for each row will tell us how many hours the person has worked each week. The totals for each column will tell us how many hours the person worked on Mondays versus Tuesdays versus Wednesdays, and so on.

It's clear that the number 7 is likely to show up in several places in this program, so it makes sense to define a class constant that we can use instead of the magic number:

```
public static final int DAYS = 7;  // # of days in a week
```

Our main method can be fairly short, opening the input file we want to read and calling a method to process the file:

```
public static void main(String[] args) throws FileNotFoundException {
    Scanner input = new Scanner(new File("hours.txt"));
    processFile(input);
}
```

We saw in chapter 6 that for line-oriented data, we can generally use the following pattern as a starting point for file processing code:

```
while (input.hasNextLine()) {
    String text = input.nextLine();
    process text.
}
```

This loop allows us to process one line at a time. But each line has internal structure because there are up to 7 numbers on each line. So in this case it makes sense to construct a Scanner for each input line and to write a separate method for processing a line of data:

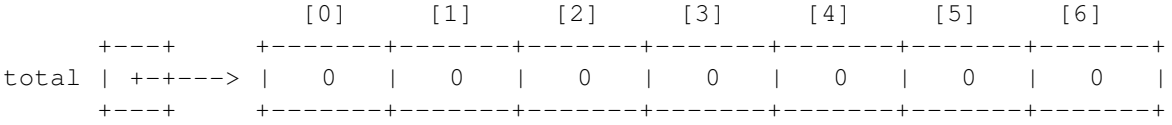
```
while (input.hasNextLine()) {
    String text = input.nextLine();
    Scanner data = new Scanner(text);
    call method to process data.
}
```

So what are we supposed to do to process a line of data? One thing we have to do is to add up the hours worked during the week and report that number. If that's all we had to do, then we could solve this as a simple file-processing problem and avoid arrays altogether. But we also have to add up columns, reporting the total hours worked on Monday, Tuesday, and so on.

This is a perfect application for an array. We are, in effect, solving seven different cumulative sum tasks. We want to sum up all hours worked on Monday, all hours worked on Tuesday, and so on. So having an array of 7 integers will allow us to calculate these column totals. So our pseudocode becomes:

```
int[] total = new int[DAYS];
while (input.hasNextLine()) {
    String text = input.nextLine();
    Scanner data = new Scanner(text);
    call method to process data.
    update total to include this week's data.
}
report total.
```

Remember that in cumulative sum we always initialize our sum to 0. When we construct an int array, Java initializes the variables to 0. So our total array will store 7 integers each with value 0:



So how do we process the data from an input line in such a way that we can report the sum across and also add the values into this total array? The first line of the sample input file is `"8 8 8 8 8"`. There are several approaches we could take, but if we're going to be manipulating an array to keep track of the total for each column, then it makes sense to create a second array that represents the data for the current week. In effect, we transfer the data from the input file into an array. Once we have the data in array form, we can do all sorts of manipulations on it.

In our pseudocode above we indicate that we should "call a method to process data" (the Scanner that contains the next line's worth of data). So let's have that method transfer the data from the Scanner into an array. It should take the Scanner as a parameter and should return a reference to a new array that it constructs. Thus, our pseudocode becomes:

```
int[] total = new int[DAYS];
while (input.hasNextLine()) {
    String text = input.nextLine();
    Scanner data = new Scanner(text);
    int[] next = transferFrom(data);
    do any processing necessary on next.
    update total to include this week's data.
}
report total.
```

The code necessary to transfer data from the Scanner into an array is not particularly complex, but it makes sense to separate this kind of operation into another method to keep the file processing method short and easy to read. We will write the code for transferFrom shortly, but we are actually fairly close to finishing this pseudocode.

We have just three pieces left to fill in. Once we have transfered the data from an input line into a Scanner, what do we do with it? We have to report the total hours for this particular week and we have to add this week's hours into our cumulative sum for the columns. Neither of these operations is particularly complex, but we can again simplify our file-processing code by introducing methods

that perform most of the details associated with these tasks. We might often find ourselves wanting to add up the numbers in an array, so it makes sense to have a method called sum that we can use to add up this row. We might also find ourselves wanting to add one array to another (the array equivalent of saying sum += next), so it also makes sense to make that a separate method.

The final element to fill in for our pseudocode is reporting the total. This is something that also can be easily put in a separate method. With these three methods in mind, we can finish translating our pseudocode into actual code:

```
int[] total = new int[DAYS];
while (input.hasNextLine()) {
    String text = input.nextLine();
    Scanner data = new Scanner(text);
    int[] next = transferFrom(data);
    System.out.println("Total hours = " + sum(next));
    addTo(total, next);
}
System.out.println();
print(total);
```

The transferFrom Method

The transferFrom method is given a Scanner as a parameter and is supposed to construct a new array that has the numbers from the Scanner. Each input line has at most 7 numbers, but it might have fewer. As a result, it makes sense to use a while loop that tests whether there are more numbers left to read from the Scanner.

```
construct an array of 7 integers.
while (data.hasNextInt()) {
    process data.nextInt().
}
```

In this case, processing the next integer from the input means storing it in the array. The first number should go into position 0, the second number in position 1, the third number in position 2 and so on. That means that we need some kind of integer counter that goes up by one each time through the loop.

```
construct an array of 7 integers.
start i at 0.
while (data.hasNextInt()) {
    store data.nextInt() in position i of the array.
    increment i.
}
```

This is now fairly easy to translate into actual code.

```
int[] result = new int[DAYS];
int i = 0;
while (data.hasNextInt()) {
    result[i] = data.nextInt();
    i++;
}
return result;
```

It may seem odd to have a method return an array, but it makes perfect sense to do so in Java. Arrays are objects, so what is returned is a reference to the array. We have to be careful to specify this properly in our header. We are returning something of type int[], so the header would look like this:

```
public static int[] transferFrom(Scanner data)
```

The sum Method

This is the simplest of the four array-processing methods that we need to write. We simply want to add up the numbers stored in the array. This is a classic cumulative sum problem and we can use a for-each loop to accomplish the task.

```
public static int sum(int[] numbers) {
    int sum = 0;
    for (int n : numbers) {
        sum += n;
    }
    return sum;
}
```

The addTo Method

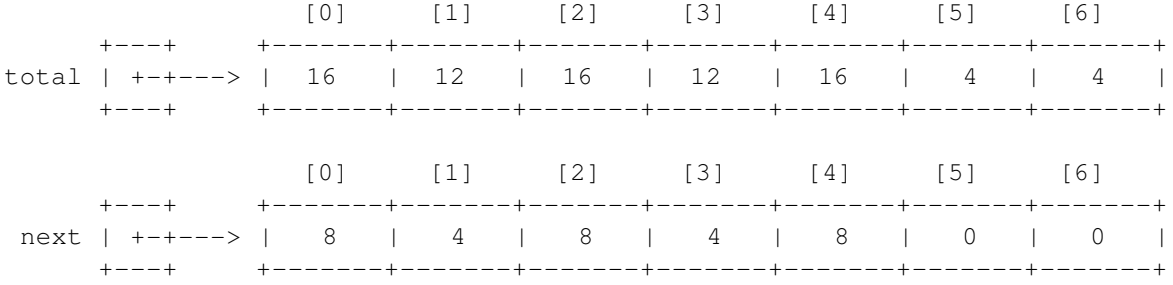
The idea for this method is to write the array equivalent of:

```
sum += next;
```

We will be given two arrays, one with the total hours and one with the next week's hours. Let's consider where we'll be in the middle of processing the sample input file. The first three lines of input are as follows:

```
8 8 8 8 8
8 4 8 4 8 4 4
8 4 8 4 8
```

Suppose we have properly processed the first two lines and have just read in the third line for processing. Then our two arrays will look like this:



It would be nice if we could just say:

```
total += next;
```

Unfortunately, you can't use operations like + and += on arrays. But those operations can be performed on simple integers and these arrays are composed of simple integers. So we basically just have to tell the computer to do 7 different += operations on the individual array elements. This can be easily written with a for loop.

```
public static void addTo(int[] total, int[] next) {
    for (int i = 0; i < DAYS; i++) {
        total[i] += next[i];
    }
}
```

The print Method

The final method we have to write involves printing out the column totals (the totals for each day of the week). The basic code involves a simple for loop, as in:

```
for (int i = 0; i < DAYS; i++) {
    System.out.println(total[i]);
}
```

But we don't want to simply write out 7 lines of output each with a number on it. It would be nice to give some information about what the numbers mean. We know that total[0] represents the total hours worked on various Mondays and total[1] represents the total hours worked on Tuesdays, and so on, but somebody reading our output might not know. So it would be helpful to label the output with some information about which day each total goes with. We can do so by defining our own array of String literals:

```
String[] dayNames = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
```

Using this array, we can improve our for loop to print out the name of each day along with its total. We can also call the sum method to write out the overall total hours worked after the for loop.

```
public static void print(int[] total) {
    String[] dayNames = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
    for (int i = 0; i < DAYS; i++) {
        System.out.println(dayNames[i] + " hours = " + total[i]);
    }
    System.out.println("Total hours = " + sum(total));
}
```

The Complete Program

Putting all the pieces together, we end up with the following complete program.

```
1  // This program reads an input file with information about hours worked and
2  // produces a list of total hours worked each week and total hours worked for
3  // each day of the week.  Each line of the input file should contain
4  // information for one week and should have up to 7 integers representing how
5  // many hours were worked each day (Monday first, then Tuesday, through
6  // Sunday).
7
8  import java.io.*;
9  import java.util.*;
10
11 public class Hours {
12     public static final int DAYS = 7;  // # of days in a week
13
14     public static void main(String[] args) throws FileNotFoundException {
15         Scanner input = new Scanner(new File("hours.txt"));
16         processFile(input);
17     }
18
19     // processes an input file of data about hours worked by an employee
20     public static void processFile(Scanner input) {
21         int[] total = new int[DAYS];
22         while (input.hasNextLine()) {
23             String text = input.nextLine();
24             Scanner data = new Scanner(text);
25             int[] next = transferFrom(data);
26             System.out.println("Total hours = " + sum(next));
27             addTo(total, next);
28         }
29         System.out.println();
30         print(total);
31     }
32
33     // constructs an array of DAYS integers initialized to 0 and transfers
34     // data from the given Scanner into the array in order; assumes the
35     // Scanner has no more than DAYS integers
36     public static int[] transferFrom(Scanner data) {
37         int[] result = new int[DAYS];
38         int i = 0;
39         while (data.hasNextInt()) {
40             result[i] = data.nextInt();
41             i++;
42         }
43         return result;
44     }
45
46     // returns the sum of the integers in the given array
47     public static int sum(int[] numbers) {
48         int sum = 0;
49         for (int n : numbers) {
50             sum += n;
51         }
52         return sum;
53     }
54
55     // adds the values in next to their corresponding entries in total
56     public static void addTo(int[] total, int[] next) {
57         for (int i = 0; i < DAYS; i++) {
58             total[i] += next[i];
59         }
60     }
```

```
61
62      // prints information about the totals including total hours for each
63      // day of the week and total hours overall
64      public static void print(int[] total) {
65          String[] dayNames = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
66          for (int i = 0; i < DAYS; i++) {
67              System.out.println(dayNames[i] + " hours = " + total[i]);
68          }
69          System.out.println("Total hours = " + sum(total));
70      }
71  }
```

When executed on the sample input file:

```
8 8 8 8 8
8 4 8 4 8 4 4
8 4 8 4 8
3 0 0 8 6 4 4
8 8
0 0 8 8
8 8 4 8 4
```

It produces the following output:

```
Total hours = 40
Total hours = 40
Total hours = 32
Total hours = 25
Total hours = 16
Total hours = 16
Total hours = 32

Mon hours = 43
Tue hours = 32
Wed hours = 36
Thu hours = 40
Fri hours = 34
Sat hours = 8
Sun hours = 8
Total hours = 201
```

Chapter Summary

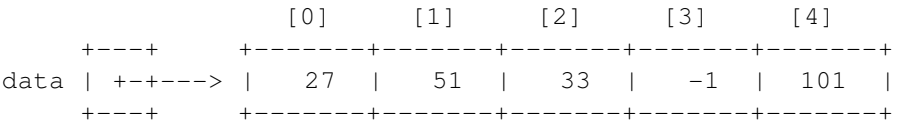
- An array is an object that groups many primitive values or objects by one name. Each individual value, called an element, is accessed using an integer index beginning at 0. Arrays are declared with square brackets, such as `int[] a = new int[5];` . An array's elements are indexed from 0 to one less than the array's length.
- An array element's value may be accessed or modified by writing the array's name and the element's index in square brackets, such as `a[2]` to refer to the third element (index 2) in the array a.
- Arrays are often traversed using for loops. The length of an array is found by accessing its `length` field, so the loop over an array a can run from indexes 0 to a.length - 1.

- Attempting to access an array element with an index less than 0 or greater than or equal to the array's length will cause the program to crash with an ArrayIndexOutOfBoundsException.
- Arrays have a fixed size. If we find that we want a larger array, we have to create a new array of the larger size and copy the old array's contents into the new array. Arrays do not support common operations like == for comparison, nor can an array's elements be printed using System.out.println. To achieve such bulk operations, you must explicitly write loops to print or compare each element of the array.
- A shortcut syntax for initializing array elements is to write the elements between { } braces when declaring the array, such as: `int[] numbers = {42, 17, 26, -5, 2};` The array will be created with such a length that it exactly fits the elements written (in this case, 5).
- The Arrays class in the java.util library package contains several useful methods for manipulating arrays, such as `Arrays.sort`, which rearranges the elements of the given array into ascending order.
- Arrays of objects are actually arrays of references to objects. A newly declared and initialized array of objects actually stores `null` in all of its element indexes, so each element must be initialized individually or in a loop to store an actual object.
- A multidimensional array is an array of arrays. These are often used to store two-dimensional data as rows and columns, or x/y data in a 2D space.

Self-Check Problems

Section 7.1: Array Basics

1. What expression should be used to access the first element of an array of integers named `numbers`? What expression should be used to access the last element of `numbers`, assuming it contains 10 elements? What expression can be used to access its last element, regardless of its length?
2. Write code that creates an array of integers named `data` of size 5 with the following contents:



3. Write code that stores all odd numbers between -6 and 38 into an array using a loop. Make the array's size exactly large enough to store the numbers.

   Try generalizing your code so that it would work for any minimum and maximum value, not just -6 and 38.

4. What elements does the array `numbers` contain after the following code?

```
int[] numbers = new int[8];
numbers[1] = 4;
numbers[4] = 99;
numbers[7] = 2;

int x = numbers[1];
numbers[x] = 44;
numbers[numbers[7]] = 11;  // uses numbers[7] as index
```

5. Write code that computes the sum of all integers in an array. For example, if the array contains the values {74, 85, 102, 99, 101, 56, 84}, your sum should return 601. You may wish to put this code into a method named `sumAll` that accepts an array of integers as its parameter and returns the sum.

6. Write code that computes the average (arithmetic mean) of all elements in an array of integers as a double. For example, if the array passed contains the values {1, -2, 4, -4, 9, -6, 16, -8, 25, -10}, your average should be 2.5. You may wish to put this code into a method named `average` that accepts an array of integers as its parameter and returns the average.

7. What is wrong with the following code?

```
int[] first = new int[2];
first[0] = 3;
first[1] = 7;
int[] second = new int[2];
second[0] = 3;
second[1] = 7;

// print the array elements
System.out.println(first);
System.out.println(second);

// see if the elements are the same
if (first == second) {
    System.out.println("They contain the same elements.");
} else {
    System.out.println("The elements are different.");
}
```

8. Write a piece of code that examines two arrays of integers as its arguments and reports whether the two arrays contain the same elements. By definition, an array is always equal to itself. Note that arrays of different lengths cannot be equal, so the method should return false in such cases. Consider putting your code into a method named `equal` that accepts the two arrays as parameters and returns true if they are equal.

Section 7.2: Advanced Arrays

9. Write a piece of code that declares an array named `data` with the elements 7, -1, 13, 24, and 6. Use only one statement to initialize the array.

10. What are the values of the elements in the array `numbers` after the following code?

```
int[] numbers = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
for (int i = 0; i < 9; i++) {
    numbers[i] = numbers[i + 1];
}
```

11. What are the values of the elements in the array `numbers` after the following code?

```
int[] numbers = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
for (int i = 1; i < 10; i++) {
    numbers[i] = numbers[i - 1];
}
```

12. Write code that prints each element of an array of 5 integers named `data` as follows:

```
element [0] is 14
element [1] is 5
element [2] is 27
element [3] is -3
element [4] is 2598
```

    For added challenge, generalize your code so that it would work on an array of any size.

13. Consider the following method mystery:

```
public static void mystery(int[] a, int[] b) {
    for (int i = 0; i < a.length; i++) {
        a[i] += b[b.length - 1 - i];
    }
}
```

    What are the values of the elements in array `a1` after the following code executes?

```
int[] a1 = {1, 3, 5, 7, 9};
int[] a2 = {1, 4, 9, 16, 25};
mystery(a1, a2);
```

14. Consider the following method mystery:

```
public static void mystery2(int[] a, int[] b) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[2 * i % a.length] - b[3 * i % b.length];
    }
}
```

    What are the values of the elements in array `a1` after the following code executes?

```
int[] a1 = {2, 4, 6, 8, 10, 12, 14, 16};
int[] a2 = {1, 1, 2, 3,  5,  8, 13, 21};
mystery2(a1, a2);
```

15. Consider the following method mystery:

```
public static void mystery3(int[] data, int x, int y) {
    data[data[x]] = data[y];
    data[y] = x;
}
```

What are the values of the elements in the array __ after the following code executes?

```
int[] numbers = {3, 7, 1, 0, 25, 4, 18, -1, 5};
mystery3(numbers, 3, 1);
mystery3(numbers, 5, 6);
mystery3(numbers, 8, 4);
```

16. Write a piece of code that computes the average String length of the elements of an array of Strings. For example, if the array contains {"belt", "hat", "jelly", "bubble gum"}, the average length is 5.5.

17. Write code that accepts an array of Strings as its argument and indicates whether that array is a palindrome, that is, the same forwards as backwards. For example, the array {"alpha", "beta", "gamma", "delta", "gamma", "beta", "alpha"} is a palindrome.

Exercises

1. Write a method named `printAll` that accepts an array of integers as its argument and prints the array's contents on one line. For example, if the array variable contains the elements 74, 85, 102, 99, 101, 56, and 84, your method should produce the following output:

   ```
   {74, 85, 102, 99, 101, 56, 84}
   ```

2. Write a method named `swap` that accepts an array of integers and two integer indexes as its arguments, and swaps the elements at those indexes in the array. For example, if an array named `numbers` contains the elements {13, 27, 92, -6, 45}, the call `swap(numbers, 1, 3);` would cause the array to contain the elements {13, -6, 92, 27, 45}. You may assume that the indexes passed to your `swap` method are within the bounds of the array. You may wish to use the `printAll` method from the previous exercise to print the array's contents to check your answer.

3. Write a method named `reverse` that accepts an array of integers as its argument and reverses the order of the elements in that array. For example, if the array {13, 27, 92, -6, 45} is passed to your `reverse` method, afterward its contents should be {45, -6, 92, 27, 13}. Your method should work on an array of any length. Note that you may have to deal with arrays of even or odd size! HINT: You may wish to use a solution to an earlier exercise to help you solve this problem.

4. Write a method named `copyAll` that accepts two arrays of integers as its arguments and copies the element values from the first array into the second array. For example, consider the following code:

   ```
   int[] array1 = {2, 1, 4, 3, 6, 5};
   int[] array2 = new int[6];
   copyAll(array1, array2);
   ```

   After your `copyAll` method executes, the array `array2` should have the contents {2, 1, 4, 3, 6, 5} just like `array1`. You may assume that the second array is always at least as large in length as the first array.

Also try writing a variation of this method that only accepts one array as a parameter and returns the copied array. Inside the method, create a second array of numbers to hold the copy and return it.

5. Write a method named `copyRange` that takes two arrays *a1* and *a2*, two starting indexes *i1* and *i2*, and a length *l*, and copies the first *l* elements of *a1* starting at index *i1* into array *a2* starting at index *i2*. Assume that the arguments' values are valid, that the arrays are large enough to hold the data, and so on.

6. Write a method named `mode` that returns the most frequently occurring element of an array of integers. Assume that the array has at least one element and that every element in the array has a value between 0 and 100 inclusive. Break ties by choosing the lower value. For example, if the array passed contains the values {27, 15, 15, 11, 27}, your method should return 15.

   Hint: You may wish to look at the tally program to get an idea how to solve this problem. Can you write a version of this method that does not rely on the elements being between 0 and 100?

7. Write a method named `stdev` that returns the standard deviation of an array of integers. Standard deviation is computed by taking the square root of the sum of the squares of the differences between each element and the mean, divided by one less than the number of elements. (It's just that simple!) More concisely and mathematically, the standard deviation of an array a is written as follows:

$$stdev(a) = \sqrt{\frac{\sum_{i=0}^{a.length-1} (a[i] - average(a))^2}{a.length - 1}}$$

   For example, if the array passed contains the values {1, -2, 4, -4, 9, -6, 16, -8, 25, -10}, your method should return approximately 11.237.

8. Write a method named `kthLargest` that accepts an integer *k* and an array *a* as its parameters and returns the element such that *k* elements have greater or equal value. If k = 0, return the largest element; if k = 1, return the 2nd largest element, and so on. For example, if the array passed contains the values {74, 85, 102, 99, 101, 56, 84} and the integer *k* passed is 2, your method should return 99, because there are 2 values at least as large as 99 (101 and 102). Assume that $0 \leq k <$ *a*.length.

   Hint: consider sorting the array (or sorting a copy of the array) first.

9. Write a method named `median` that accepts an array of integers as its argument and returns the median of the numbers in the array. The median is the number such that if you arranged the elements in order, it would appear in the middle. Assume that the array is of odd size (so that one sole element constitutes the median) and that the numbers in the array are between 0 and 99 inclusive. For example, the median of {5, 2, 4, 17, 55, 4, 3, 26, 18, 2, 17} is 5 and the median of {42, 37, 1, 97, 1, 2, 7, 42, 3, 25, 89, 15, 10, 29, 27} is 25.

   Hint: You may wish to look at the Tally program from earlier in this chapter for ideas.

10. Rewrite your `median` method so that it works regardless of the elements' values, and also works for arrays of even size. (The median of an array of even size is the average of the middle two elements.)

11. Use your `median` method from the previous exercises to write a program that reads a file full of integers and prints the median of those integers.

12. Write a method named `wordLengths` that accepts a String representing a file name as its argument. Your method should open the given file and count the number of letters in each token in the file, and output a result diagram of how many words contained each number of letters. For example, if the file contains the following text:

```
Before sorting:
13 23 480 -18 75
hello how are you feeling today

After sorting:
-18 13 23 75 480
are feeling hello how today you
```

Your method should produce the following output to the console:

```
1: 0
2: 6    ******
3: 10   **********
4: 0
5: 5    *****
6: 1    *
7: 2    **
8: 2    **
```

Assume that no token in the file is more than 80 characters in length.

Programming Projects

1. Java's type int has a limit on how large of an integer it can store. This limit can be circumvented by representing an integer as an array of digits. Write an interactive program that adds two integers of up to 50 digits each.

2. Personal mailing labels can prove quite useful. Write a program that reads a five-line address from an input file and produces an output file with the address repeated 50 times in three columns.

3. Write a program that reads an input file of numbers and reports the average of the numbers as well as how far each number is from the average.

4. Write a program that plays a variation of the game of Mastermind with a user. For example, the program can use pseudorandom numbers to generate a 4-digit number. The user should be allowed to make guesses until the user gets the number correct. Clues should be given to the user indicating how many digits of the guess are correct and in the correct place, and how many are correct but in the wrong place.

Stuart Reges
Marty Stepp

Chapter 8
Classes

Copyright © 2006 by Stuart Reges and Marty Stepp

- 8.1 Object-Oriented Programming Concepts
  - Classes and Objects
- 8.2 Object State: Fields
- 8.3 Object Behavior: Methods
  - A Detailed Example
  - Mutators and Accessors
- 8.4 Object Initialization: Constructors
- 8.5 Encapsulation
  - Private Data Fields
  - Class Invariants
- 8.6 More Instance Methods
  - The toString Method
  - The equals Method
- 8.7 The this Keyword
  - Multiple Constructors
- 8.8 Case Study: Designing a Stock Class
  - Stock Behavior
  - Stock Fields
  - Stock Constructor
  - Stock Method Implementation
  - The Complete Stock Class

Introduction

Now that we have spent time mastering the basics of procedural style programming in Java, we are finally ready to explore what Java was designed for: object-oriented programming. This chapter introduces the basic terminology we use when talking about objects and shows you how to declare your own classes to create your own objects.

We'll see that objects are entities that contain state and behavior and can be used as parts of larger programs. We'll discuss the concepts of abstraction and encapsulation, which allow us to use objects at a high level without understanding their inner details. We'll also discuss ideas for designing new classes of objects and implementing the programs that utilize them.

8.1 Object-Oriented Programming Concepts

Most of our focus so far has been on procedural decomposition, breaking a complex task into smaller subtasks. This is the oldest style of programming and even in a language like Java we still use procedural techniques. But Java provides a different approach to programming that we call *object-oriented programming*.

Object-Oriented Programming (OOP)

Reasoning about a program as a set of objects rather than as a set of actions.

Object-oriented programming involves a particular view of programming that has its own terminology. Let's explore that terminology with non-programming examples first. We've been using the word *object* for a while without giving a very good definition of the word. Below is a definition that is often used by object-oriented programmers.

Object

A programming entity that contains state and behavior.

To understand this definition, we have to understand the terms "state" and "behavior." These are some of the most fundamental concepts in object-oriented programming.

Let's consider the class of objects we call radios. What are the different states a radio can be in? It can be turned on or turned off. It can be tuned to one of many different stations and it can be set to different volumes. Any given radio has to "know" what state it is in, which means that it has to keep track of this information internally. We call the collection of such internal values the *state* of an object.

State

A set of values (internal data) stored in an object.

What are the behaviors of a radio? The most obvious one is that it produces sound when it is turned on and the volume is turned up. But it has other behaviors that involve the manipulation of its internal state. We can turn a radio on or off. We can change the station or volume. We can see what station the radio is set to right now. We call the collection of these operations the *behavior* of an object.

Behavior

A set of actions an object can perform, often reporting or modifying its internal state.

Objects themselves are not complete programs. Instead they are components that are given distinct roles and responsibilities. Objects can be used as part of larger programs to solve problems. The pieces of code that create and use objects are known as *clients*.

Client (or client code)

Code that uses an object.

Client programs interact with objects by sending messages to them, asking them to perform behavior. A major benefit of objects is that they provide reusable pieces of code that can be utilized in many client programs. You've used several interesting objects starting in Chapter 3, such as those of type String, Point, Scanner, Random, and File. Java's class libraries contain over 3,000 existing classes of objects.

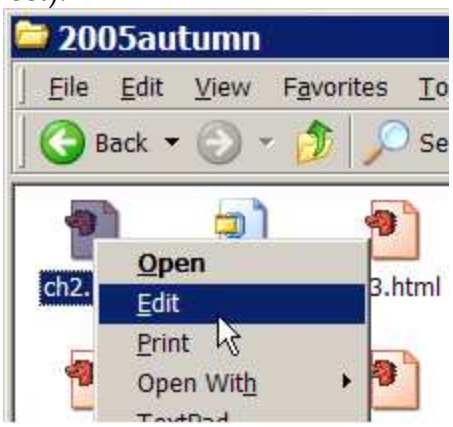
As you write larger programs you'll find cases where Java doesn't have a pre-existing object for the problem you're solving. For example, if you were creating a calendar application, you might want objects to represent dates, contacts, and appointments. If you were creating a 3-dimensional graphical simulation, you might want objects to represent 3D points, vectors, and matrices. If you were writing a financial program, you might want classes to represent your various assets, transactions, and expenses. In this chapter we'll learn how to create our own classes of objects that can be used by client programs like these.

Did You Know: Operating Systems History and Objects

To understand the basic idea of OOP, consider the history of how users interact with a computer. If you went back to 1983 you'd find that the IBM PC and its "clones" were the dominant machines and most people were running an operating system called DOS. DOS uses what we call a "command line interface" in which the user types commands at a prompt. The console window we have been using is a similar interface. To delete a file in DOS, for example, you would give the command "del" (short for "delete") followed by the file name, as in:

```
del data.txt
```

This interface can be described in simple terms as "verb noun". In fact, if you look at a DOS manual, you will find that it is full of verbs. This closely parallels the procedural approach to programming. When we want to accomplish some task, we issue a command (the verb) and then mention the object of the action (the noun, the thing we want to affect).



In 1984 Apple Computer released a new computer called a Macintosh that had a different operating system that used what we call a Graphical User Interface or GUI. The GUI interface uses a graphical "desktop" metaphor that has become so well known that people almost forget that it is a metaphor. The Macintosh was not the first computer to use the desktop metaphor and a GUI, but it was the first such computer that became a major commercial success. Later Microsoft brought this functionality to IBM PCs with its Windows operating system.

So how do you delete a file on a Macintosh or on a Windows machine? You locate the icon for the file and click on it. Then you have several options. You can drag it to the garbage/recycling bin. Or you can give a "delete" command. Either way you start with the thing you want to delete and then give the command you want to perform. This is a reversal of the fundamental paradigm. In DOS it was "verb noun" but with a GUI it's "noun verb". This different way of viewing the world is the core of object-oriented programming.

Most modern programs use a graphical user interface because we have learned that people find it more natural to work this way. We are used to pointing at things, picking up things, grabbing things. Starting with the object is very natural for us. This approach has also proved to be a helpful way to structure our programs, by dividing our programs up into different objects that each can do certain tasks rather than dividing up a task into subtasks.

Classes and Objects

In the previous chapters, we've considered the words "class" and "program" to be roughly synonymous. We wrote programs by creating a new class and placing a main static method into that class.

But classes have another use in Java: to serve as a template for a new type of objects. To create a new type of objects in Java, we must create a class and add code to it that specifies the following things:

- The state stored in each object
- The behavior each object can perform
- How to construct objects of the type

Once we have written the appropriate code, the class can be used to create objects of its type. We can then use those objects in our client programs. We say that objects are *instances* of the class because one class can be used to construct many objects.

Another way to think of a class is as a blueprint. A blueprint contains the directions to build a house. One blueprint can be used to create many similar houses, each with its own state. Similarly a class contains the directions to build objects with state and behavior.

In the next several sections we'll explore the structure of a class by writing a new class incrementally. We'll write our own version of the Point type from the java.awt package. A Point object represents a 2-dimensional (x, y) location. Point objects are useful for applications that store many 2D locations, such as maps of cities, graphical animations and games.

The main components of a class that we'll see in these sections are:

- fields (the data stored in each object),
- methods (the behavior each object can execute),
- constructors (special methods used to construct objects with the `new` keyword), and
- encapsulation (protecting an object's data from outside access).

We'll focus on these concepts by creating four major versions of the Point class. Our first version will give us Point objects that contain only data. The second version will add behavior to the objects. The third version will allow us to construct Points at any initial position. The fourth version will encapsulate each Point's internal data from unwanted outside access. Only the fourth version of the Point class will be written in proper object-oriented style; the others will be incomplete and used to illustrate each feature of classes in isolation.

8.2 Object State: Fields

The first version of our Point class will contain state only. To specify each object's state, we declare special variables inside the class called *fields*. There are many synonyms for "field" that come from other programming languages and environments, such as "instance variable," "data member," and "attribute."

Field (or Instance Variable)

A variable inside an object that makes up part of its internal state.

When creating a new class, we create a file for the class and then specify the state and behavior each object of that class should have. Data fields are declared by writing a type and name inside the braces for the class. We'll see a modified syntax for data fields later in this chapter when we discuss encapsulation.

Here's the first version of our Point class. This code creates a new class named Point and declares that each Point object will contain two fields: an integer named x and an integer named y. The class must be saved into a file named Point.java.

```
1  // A Point object represents an ordered pair of (x, y) coordinates.
2  // First version: state only.
3
4  public class Point {
5      int x;
6      int y;
7  }
```

Though the syntax for data fields makes them look like normal variables, they are very different. The preceding code is not just declaring a pair of int variables named x and y. The code is actually declaring that every Point object will contain its own int fields named x and y. If we create one hundred Point objects, we'll have one hundred pairs of x and y fields, one held inside each object.

One thing you'll encounter when you're writing classes of objects and client code is that your program will now occupy more than one file. The Point class is stored in Point.java, and the client code that uses Point exists in a separate file. The Point.java file isn't an executable Java program in itself, but simply a definition of a new class of objects for client programs to use. Point.java and its client programs should be in the same folder on your computer so that the classes can find each other and compile successfully.

For now, client programs create Point objects with the `new` keyword and empty parentheses, such as:

```
Point origin = new Point();
```

Recall that a Point object is also called an instance of the Point class. The Point class serves as the blueprint for the contents of each of its instances. Client programs can create as many Point instances as needed.

When a Point object is initially constructed, its data fields are given default initial values of 0, so a new Point object always begins at the origin of (0, 0) until we change its x/y values. This automatic initialization is similar to the way that array elements are automatically given default values.

Here's the first version of a client program that uses our Point class. It would be saved into a separate file named PointMain.java.

```
1  // A program that deals with 2D points.
2  // First version, to accompany Point class with state only.
3
4  public class PointMain {
5      public static void main(String[] args) {
6          // create two Point objects
7          Point p1 = new Point();
8          p1.x = 7;
9          p1.y = 2;
10
11         Point p2 = new Point();
12         p2.x = 4;
13         p2.y = 3;
14
15         // print each point and its distance from origin
16         System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
17         double dist1 = Math.sqrt(p1.x * p1.x + p1.y * p1.y);
18         System.out.println("distance from origin = " + dist1);
19
20         System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
21         double dist2 = Math.sqrt(p2.x * p2.x + p2.y * p2.y);
22         System.out.println("distance from origin = " + dist2);
23         System.out.println();
24
25         // translate each point to a new location
26         p1.x += 11;
27         p1.y += 6;
28         p2.x += 1;
29         p2.y += 7;
30
31         // print the points again
32         System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
33         System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
34     }
35 }
```

The code produces the following output:

```
p1 is (7, 2)
distance from origin = 7.280109889280518
p2 is (4, 3)
distance from origin = 5.0

p1 is (18, 8)
p2 is (5, 10)
```

The client program has some redundancy that we'll eliminate as we improve our Point class in the next sections.

8.3 Object Behavior: Methods

The second version of our Point class will contain both state and behavior. Behavior of objects is specified by writing *instance methods*. The instance methods of an object describe what messages that object can receive.

Instance Method

A method that exists inside an object and operates on that object, making up part of its behavior.

An instance method exists inside of an object and is called on that particular object by writing the object variable's name, a dot, and the method name. In previous chapters you have already called instance methods on several types of Java objects, such as the length method of a String or the nextInt method of a Scanner.

Our client program from the previous section translates the position of two Point objects. Our Point type doesn't have any behavior yet, so the client program is forced to translate points manually by adjusting their x/y field values.

```
p1.x += 11;   // our Point doesn't have a translate method
p1.y += 6;
```

Translating points is a common operation, so we should represent it as a method. You might be tempted to write a static translate method in your client code that accepts a Point, a delta-x, and a delta-y as parameters. Its code would look like this:

```
// A static method to translate a Point.
// Not a good model to follow.
public static void translate(Point p, int dx, int dy) {
    p.x += dx;
    p.y += dy;
}
```

A call to the static method would look like this:

```
translate(p1, 11, 6);   // calling a translate static method
```

But a static method isn't the best way to implement the translate behavior. One of the biggest benefits of programming with objects is that we can put related data and behavior together. The ability for a Point to translate is closely related to that Point's x/y data, so it's better to specify that each Point object will know how to translate itself. We'll do this by writing an instance method in the Point class.

The client program could call a Point's translate method like the following call. Notice that our instance method only needs two parameters: dx and dy. The client doesn't pass the Point as a parameter; instead it sends a translate message to the Point object by writing the object reference's name, `p1`, before the dot and method name.

```
p1.translate(11, 6);   // calling a translate instance method
```

Instance method headers do not have the `static` keyword that static methods have. We still include the public keyword, the method's return type, its name, and any parameters the method accepts. Here's the start of a Point class with a translate method. We've declared the header but left the body blank for a moment.

```
public class Point {
    int x;
    int y;

    public void translate(int dx, int dy) {
        <code to implement the method>
    }
}
```

Remember that when we declare a translate method in the Point class, we are saying that each Point object has its own copy of that method. So when we're writing the body of the translate method, we have to think from the perspective of a particular Point object: "The client has given me a dx and dy, and she wants me to change my x and y values by those amounts."

In previous chapters we've talked about scope, the range in which a variable can be seen and used. The scope of a data field spans its entire class. This means that a Point object's instance methods can refer to its data fields simply by writing their name. So if for example the Point wants to adjust its x value by dx, we can simply write the statement `x += dx;` . Here is a working translate method that adjusts the Point's position:

```
public void translate(int dx, int dy) {
    x += dx;
    y += dy;
}
```

It might seem strange that the translate method can refer to x and y directly without being more specific about which object it's affecting. As a non-programming analogy, if you're standing outside near several cars, you might point at one and say, "Turn that car's steering wheel to the left." But if you were inside the car, you'd simply say, "Turn left." You wouldn't feel a need to specify which car, because it's implied that you want to steer the car you're currently occupying. In instance methods, we don't need to specify which object's x or y we're using, because it's implied that we want to use the one of the object that received the message.

Here's the complete Point class that contains a translate method. Java style guidelines suggest to declare fields at the top of the class, with methods below. But in general it is legal for a class's contents to appear in any order.

```
// A Point object represents an ordered pair of (x, y) coordinates.
// Second version: state and behavior.

public class Point {
    int x;
    int y;

    // Shifts this Point's position by the given amount.
    public void translate(int dx, int dy) {
        x += dx;
        y += dy;
    }
}
```

The general syntax for instance methods is the following:

```
public <type> <name>(<type> <name>, ..., <type> <name>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

We said a moment ago that an instance method's body has an implicit knowledge of what object it's operating upon. This implicit knowledge is sometimes called the *implicit parameter*. Like a parameter, it is a piece of information that changes how a method executes.

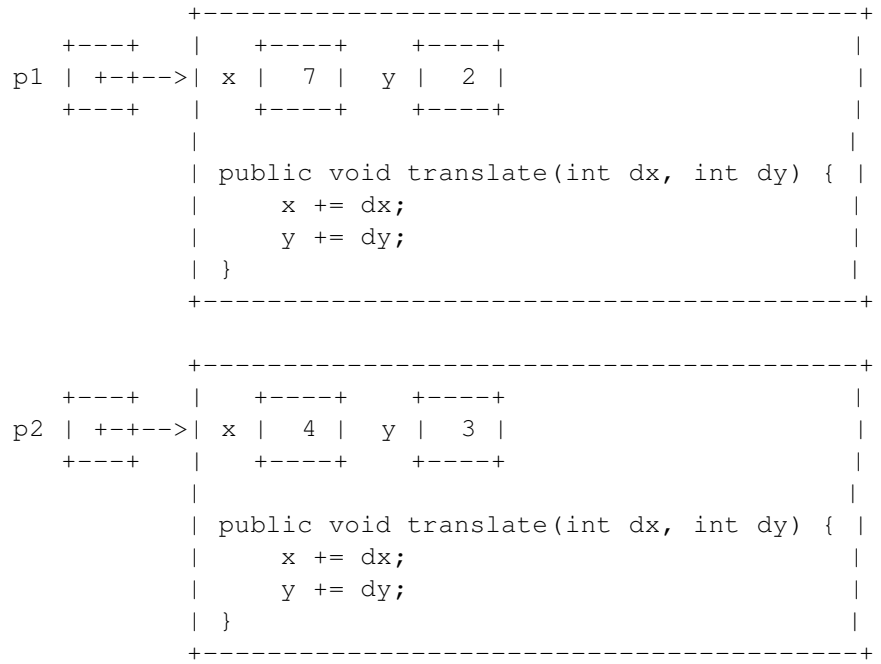
> **Implicit Parameter**
>
> The object being acted upon in an instance method.

A Detailed Example

Let's walk through an example in detail to understand how instance methods work. Here we'll write some client code that constructs two Point objects and sets initial positions for them.

```
// Construct two Point objects
Point p1 = new Point();
p1.x = 7;
p1.y = 2;

Point p2 = new Point();
p2.x = 4;
p2.y = 3;
```

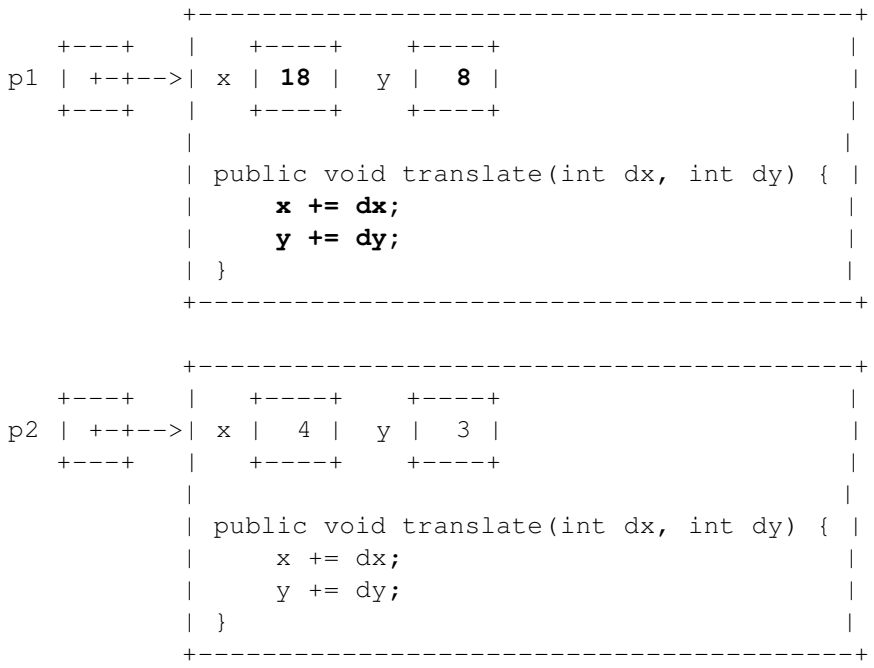
After the preceding code, a diagram of the variables and objects in memory would look like this. Remember that each object has its own copy of the translate instance method.



Now we'll call the translate method on each object. First, p1 is translated.

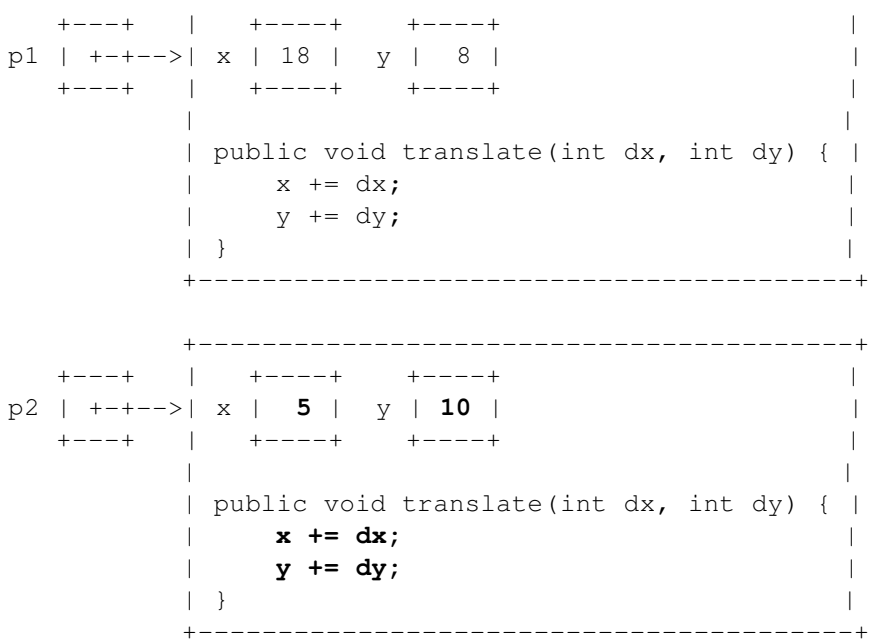
```
p1.translate(11, 6);
```

When an instance method is called on p1, p1's copy of the translate method is passed the parameters 11 and 6 and is executed. So the statements `x += dx;` and `y += dy;` affect the x and y data fields inside p1's object.



During the second method call, p2's copy of the method is executed. So the lines in the body of the translate method change the x and y fields of p2's object.

```
p2.translate(1, 7);
```



Methods like translate are useful because they give our objects useful behavior that lets us write more expressive and concise client programs. Having the client code manually adjust the x and y values of Point objects to move them is tedious, especially in larger client programs that translate many Points. By adding the translate method, we have given a clean way to adjust the position of a Point in a single statement.

Mutators and Accessors

Our translate method is an example of a kind of instance methods informally known as mutators. A *mutator* is an instance method that changes the state of that object in some way. Generally this means that a mutator assigns a new value to one of the object's data fields. For a radio, the mutators would be switches and knobs that turned the radio on and off or that changed the station or that changed the volume. You have already used several mutators, such as the setLocation of Java's Point objects or the nextInt method on Scanner objects.

Mutator

A method of an object that modifies its internal state.

There are some conventions about how to write mutators. Many mutator methods' names begin with "set", such as setID or setTitle. Usually mutator methods have a void return type. Mutators often accept parameters that specify the new state of the object, or the amount by which to modify its current state.

A second important category of instance methods is known as accessors. An *accessor* is an instance method that provides information about the state of that object. For example, a radio object might provide access to its current station setting or current volume. Examples of accessor methods you have seen include the length and substring methods of String objects, or the isDirectory method on File objects.

Accessor

An instance method that lets client code read the object's internal state.

Our client program computes how far two Points are from the origin of (0, 0). Since this is a common operation related to the data in a Point, let's give each Point an accessor named distanceFromOrigin that computes and returns that Point's distance from the origin. The method accepts no parameters and returns the distance as a double.

The distance from the origin is computed using the Pythagorean Theorem, taking the square root of the sum of the squares of the x and y values. As in the translate method, we'll refer to the Point's x and y fields directly in our computation.

```
// Returns the direct distance between this Point and the origin (0, 0).
public double distanceFromOrigin() {
    return Math.sqrt(x * x + y * y);
}
```

Note that the distanceFromOrigin method doesn't change the Point's x or y values. Accessors are not used to change the state of the object--they only report information about the object.

There are some conventions about how to write accessors. They usually don't need to accept any parameters, and they usually have a non-void return type to allow them to return the appropriate information. Many accessors' names begin with the prefix "get" or "is", such as getBalance or isEmpty.

Here's the complete second version of our Point class that now contains both state and behavior.

```
1  // A Point object represents an ordered pair of (x, y) coordinates.
2  // Second version: state and behavior.
3
4  public class Point {
5      int x;
6      int y;
7
8      // Returns the direct distance between this Point and the origin (0, 0).
9      public double distanceFromOrigin() {
10         return Math.sqrt(x * x + y * y);
11     }
12
13     // Shifts this Point's position by the given amount.
14     public void translate(int dx, int dy) {
15         x += dx;
16         y += dy;
17     }
18 }
```

The client program can now use the new behavior of the Point class. The resulting program produces the same output as before but is shorter and more readable than the original.

```
1  // A program that deals with 2D points.
2  // Second version, to accompany Point class with state and behavior.
3
4  public class PointMain {
5      public static void main(String[] args) {
6          // create two Point objects
7          Point p1 = new Point();
8          p1.x = 7;
9          p1.y = 2;
10
11         Point p2 = new Point();
12         p2.x = 4;
13         p2.y = 3;
14
15         // print each point and its distance from origin
16         System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
17         System.out.println("distance from origin = " + p1.distanceFromOrigin());
18
19         System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
20         System.out.println("distance from origin = " + p2.distanceFromOrigin());
21
22         // translate each point to a new location
23         p1.translate(11, 6);
24         p2.translate(1, 7);
25
26         // print the points again
27         System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
28         System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
29     }
30 }
```

8.4 Object Initialization: Constructors

Our third version of the Point class will contain the ability to create Point objects at any initial position. The initial state of objects is specified by writing a special method called a *constructor*.

> **Constructor**
>
> A method that creates a new object and initializes its state.

An annoying problem in our client code so far is that it takes three lines to create and initialize the state of one Point object.

```
// client needs 3 statements to initialize one Point object (bad)
Point p1 = new Point();
p1.x = 7;
p1.y = 2;
```

You may recall that in Chapter 3, we were able to create Java Point objects with initial positions in a single statement. We did this by writing their initial x/y values in parentheses as we constructed the `new Point`.

```
Point p = new Point(10, 27);   // this is legal with Java's Point type
```

Such a statement wouldn't be legal for our Point class, because we haven't written any code specifying how to create a Point with an initial (x, y) position. We can specify how to do this by writing a constructor in our Point class. A constructor is the piece of code that is executed when the client uses the `new` keyword to create a new object.

The code for a constructor looks like a method that has the same name as the class and has no return type specified. A constructor may accept parameters that help to initialize the object's state. Our constructor for Points will accept initial x and y values as parameters and store them into the new Point's x and y data fields. Here's the constructor code:

```
// Constructs a new Point at the given initial x/y position.
public Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
}
```

Like instance methods, constructors execute on a particular object (the one that's being created with the `new` keyword) and can refer to that object's data fields and methods directly. In our case, we store initialX and initialY parameter values into our Point's fields x and y. Here is the complete code for the third version of our Point class:

```
 1  // A Point object represents an ordered pair of (x, y) coordinates.
 2  // Third version: state and behavior with constructor.
 3
 4  public class Point {
 5      int x;
 6      int y;
 7
 8      // Constructs a new Point at the given initial x/y position.
 9      public Point(int initialX, int initialY) {
10          x = initialX;
11          y = initialY;
12      }
13
14      // Returns the direct distance between this Point and the origin (0, 0).
15      public double distanceFromOrigin() {
16          return Math.sqrt(x * x + y * y);
17      }
18
19      // Shifts the position of this Point object by the given amount
20      // in each direction.
21      public void translate(int dx, int dy) {
22          x += dx;
23          y += dy;
24      }
25  }
```

The general syntax for constructors is the following:

```
public <class name>(<type> <name>, ..., <type> <name>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

After adding the constructor, our client code becomes shorter and simpler, because it can create a Point and initialize its x/y coordinates in a single line.

```
1  // A program that deals with 2D points.
2  // Third version, to accompany Point class with constructor.
3
4  public class PointMain {
5      public static void main(String[] args) {
6          // create two Point objects
7          Point p1 = new Point(7, 2);   // calling the constructor
8          Point p2 = new Point(4, 3);   // calling the constructor
9
10         // print each point and its distance from origin
11         System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
12         System.out.println("distance from origin = " + p1.distanceFromOrigin());
13
14         System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
15         System.out.println("distance from origin = " + p2.distanceFromOrigin());
16
17         // translate each point to a new location
18         p1.translate(11, 6);
19         p2.translate(1, 7);
20
21         // print the points again
22         System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
23         System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
24     }
25 }
```

When a class doesn't have a constructor, like our Point class before this section, Java automatically supplies an empty default constructor that accepts no parameters. That is why it was previously legal to construct a `new Point()`. However, when we write a constructor of our own, Java doesn't supply the default empty constructor. So it would now be illegal to construct Point objects without passing in the initial x and y parameters.

```
Point p1 = new Point();  // this will not compile, for now
```

In a later section of this chapter, we'll write additional code to restore this ability.

Common Programming Error: Writing void on Constructor

Many new programmers accidentally write the keyword `void` on their constructor, since they've gotten so used to writing a return type on every method.

```
// This code has a bug.
public void Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
}
```

This is actually a very tricky and annoying bug. Constructors aren't supposed to have any return type written on their headers. When you do write a return type such as void, what you've created is not a constructor but rather a normal instance method, whose name is Point, which accepts x and y parameters, and returns void. This is tough to catch, because the Point.java file allows this and still compiles successfully.

The place where you will see an error is when you try to call your constructor you thought you just wrote but actually isn't a constructor. The client code that tries to construct the Point object will complain that it can't find an (int, int) constructor for a Point.

```
PointMain.java:7: cannot find symbol
symbol  : constructor Point(int,int)
location: class Point
        Point p1 = new Point(7, 2);
```

If you see "cannot find symbol" constructor errors and you were positive that you wrote a constructor, double-check its header to make sure there's no return type.

Common Programming Error: Redeclaring Data Fields in Constructor

Another common bug with constructors is to mistakenly redeclare data fields by writing their types. Here's an example that shows the mistake:

```
// This constructor code has a bug.
public Point(int initialX, int initialY) {
    int x = initialX;
    int y = initialY;
}
```

The preceding code has a very bizarre behavior. It compiles successfully, but when the client code constructs a Point object, its initial coordinates are always (0, 0) regardless of what parameter values are passed to the constructor.

```
// This client code will print that p1 is (0, 0) !
Point p1 = new Point(7, 2);
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
```

The problem is that rather than storing initialX and initialY into the Point's data fields x and y, we've actually declared local variables named x and y inside the Point's constructor. We store initialX and initialY into those local variables, which are thrown away when the constructor finishes running. No values are ever assigned to the x and y data fields in the constructor, so they receive automatic initialization to 0. We say that these local x and y variables "shadow" our x and y data fields because they obscure the fields we intended to set.

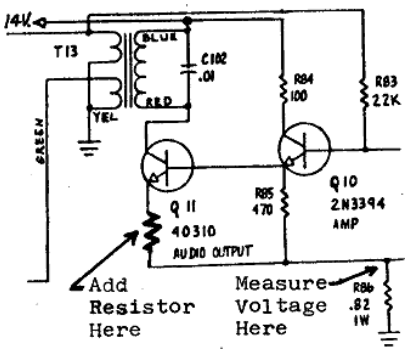
If you observe that your constructor doesn't seem to be setting your object's data fields, check closely to make sure that you didn't accidentally declare local variables that shadow your data fields. The key thing to look for is that you didn't rewrite your fields' types when trying to assign them in the constructor.

8.5 Encapsulation

Our fourth version of the Point class will take advantage of a concept known as encapsulation. To understand the notion of encapsulation, recall the non-programming analogy of radios as objects. Almost everyone knows how to use a radio, but few know how to build a radio or how the circuitry inside a radio works. It is a benefit of the radio's design that we don't need to know those kinds of details.

We all understand the essential properties of a radio (what it does), but only a few of us understand the internal details of the radio (how it works). This leads to an important dichotomy of an external view of an object versus an internal view of an object. From the outside, we just see behavior. From the inside, we see internal state that is used to accomplish that behavior.

The internal and external views of a radio.



radio, internal view

radio, external view

By focusing on the radio's external behavior, we can use it easily while ignoring its details that are unimportant to us. This is an example of an important computer science concept known as *abstraction*.

Abstraction

Focusing on an item's essential properties while ignoring its inner details.

In fact, radios and other electronic devices have a case or chassis that houses all of the electronics so that we don't even see them from the outside. Instead, we have a set of dials and buttons and displays that allow us to manipulate the radio without having to deal with all of the circuitry that makes it work. In fact, you wouldn't want someone to give you a fully functional radio that had wires and capacitors hanging out of it, because those details make the interface to the radio unclean and less pleasant to use.

When applied to programming, hiding internal state from outside view is called *encapsulation*.

Encapsulation

Hiding the implementation details of an object from the clients of the object.

When objects are properly encapsulated, the clients of an object cannot directly access or modify the internal workings of the object, nor do they need to do so. Only the implementor of the class needs to know about those details.

In previous chapters you have already taken advantage of the abstraction provided by well-encapsulated objects. You have used Scanner objects to read data from the console without knowing exactly how the Scanner stored and tokenized the input data. You have used Random objects to create random numbers without knowing exactly what algorithm the random number generator used.

But so far, our Point class is not encapsulated. We've built a working radio, but its wires (its x and y data fields) are still hanging out. Using encapsulation, we'll put a casing around our Point objects so that clients will only need to use a Point's methods and not access the fields directly.

Private Data Fields

To encapsulate data fields of an object, we declare them to be private by writing the keyword `private` at the start of the declaration of each field. The fields of our Point type would be declared as follows:

```
// encapsulated data fields of Point objects
private int x;
private int y;
```

We haven't shown a syntax template yet for data fields because we wanted to show it the preferred way, with the fields encapsulated and private. The preferred syntax for declaring data fields is the following:

```
private <type> <name>;
```

Fields can also be declared with an initial value, such as:

```
private <type> <name> = <value>;
```

Declaring fields private encapsulates the state of the object, in the same way that a radio's casing protects the user from seeing the wires and circuitry inside the radio. Private fields are visible to all of the code inside the Point class (inside the Point.java file), but not anywhere else. This means that we can no longer directly refer to a Point's x or y data fields in our client code. The following client code would not compile successfully:

```
// This client code doesn't work with encapsulated points
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
```

The compiler produces error messages such as the following:

```
PointMain.java:11: x has private access in Point
PointMain.java:11: y has private access in Point
```

To preserve the functionality of our client program, we need to provide a way for client code to access a Point's data field values. We will do this by adding some new accessor methods to the Point class. If the value of an object's data field might be useful externally, it is common to write an accessor to return that value. Here are accessors that provide access to a Point's x and y fields.

```
// Returns the x-coordinate of this Point.
public int getX() {
    return x;
}

// Returns the y-coordinate of this Point.
public int getY() {
    return y;
}
```

The client code that wishes to print a Point's x and y values must be changed to the following:

```
// This code works with our encapsulated Points
System.out.println("p1 is (" + p1.getX() + ", " + p1.getY() + ")");
```

It probably seems odd that we just granted the client access to a Point's x and y data when we just finished encapsulating those same data fields to restrict access to them. But having accessors like getX and getY doesn't break the encapsulation of the object. The accessor methods actually just return a copy of the field values to the client, which lets the client see the x or y value but doesn't give the client any way to change it. In other words, these accessor methods give the client "read only" access to the state of the object.

Another benefit of encapsulating our Point objects is that we could later change the internal structure of our Point class without having to modify our client code. For example, sometimes it is useful to express 2D points in polar coordinates using a radius *r* and an angle *theta*. In this representation, a Point's (x, y) coordinates are not stored directly but can be computed as (*r* cos *theta*, *r* sin *theta*). Now that our Point class is encapsulated, we could modify our encapsulated Point to use *r* and *theta* data fields internally, then modify the getX and getY methods to compute and return the appropriate values.

Now that our Point class is encapsulated, one drawback is that it's no longer easy for the client code to set a Point to a new location. For convenience, we'll add a new mutator to our encapsulated Point class called setLocation that sets both the x and y data fields on the object to new values passed as parameters.

```
// Sets this Point's x/y coordinates to the given values.
public void setLocation(int newX, int newY) {
    x = newX;
    y = newY;
}
```

Notice that the Point class now has some redundancy between its constructor and its setLocation method. The two bodies are essentially the same, setting the Point to have new x and y coordinates. We can eliminate this redundancy by having the constructor call setLocation rather than setting the field values manually. It is legal for an object to call its own instance methods from its constructor or other instance methods. This is done by writing the name of the method you want to call followed by any parameters in parentheses. We don't have to specify which object to call the instance method upon, because it's implicit that the object wants to call the method on itself.

```
// Constructs a new Point at the given initial x/y position.
public Point(int initialX, int initialY) {
    setLocation(initialX, initialY);
}
```

There's a bit more redundancy we can eliminate using this technique. Translating a Point can be thought of as setting its location to the old location plus the dx and dy, so we can modify the translate method to call the setLocation method.

```
// Shifts the position of this Point object by the given amount
// in each direction.
public void translate(int dx, int dy) {
    setLocation(x + dx, y + dy);
}
```

Now that we've seen all the major elements of a well-encapsulated class, it's time to look at a proper syntax template for an entire class. Java's style guidelines suggest putting data fields at the top of the class, followed by constructors, followed by methods.

```
public class <class name> {
    // data fields
    private <type> <name>;
    private <type> <name>;
    ...

    // constructors
    public <class name>(<type> <name>, ..., <type> <name>) {
        <statement(s)>;
    }
    ...

    // methods
    public <type> <name>(<type> <name>, ..., <type> <name>) {
        <statement(s)>;
    }
    ...
}
```

Class Invariants

Another benefit of encapsulated objects is that we can now place constraints on the state of an object if we wish to do so. For example, suppose we want to ensure that every Point's x and y field values are non-negative. (This might be useful if Points were being used to represent coordinates on the screen, which are never negative.) The concept of a statement that is always true about an object is sometimes called a *class invariant*. A class invariant is related to the idea of preconditions and postconditions presented in Chapter 4.

> **Class Invariant**
>
> A logical statement about an object's state that is always true for the lifetime of that object.

To enforce our class invariant, we must ensure that no Point is created with a negative x or y value and also that no Point can be moved onto a negative x/y value. This means that the x/y coordinates must be checked every time the Point's constructor, translate method, or setLocation method is called.

Since we removed redundancy by having the constructor and translate method call setLocation, we can actually enforce the invariant with a single check in the setLocation method. If the new x or y value is negative, we'll throw an IllegalArgumentException to notify the client of their error.

```
// Sets this Point's x/y coordinates to the given values.
public void setLocation(int newX, int newY) {
    if (newX < 0 || newY < 0) {
        throw new IllegalArgumentException();
    }

    x = newX;
    y = newY;
}
```

If the Point class weren't encapsulated, we wouldn't be able to properly enforce our invariant. Client code would be able to make a Point's location negative by setting its x or y data fields' values directly. For example, this piece of malicious client code would succeed in setting the Point's x value to -1:

```
// If our Point objects were not encapsulated,
// this client code would break the class invariant.
Point p1 = new Point(7, 2);
p1.x = -1;
System.out.println("Haha, I set its x value to " + p1.x);
```

With encapsulation, the Point class has much better control over how it's used by its clients, so it's not possible for a misguided client program to violate our class invariant.

Here is the fourth complete version our Point class, now including encapsulation and our non-negativity class invariant. Channeling all code that sets the Point's position through the setLocation method proves useful because we only have to check for the x and y being less than 0 once.

```
1  // A Point object represents an ordered pair of (x, y) coordinates.
2  // Fourth version: encapsulated, with a class invariant that x, y >= 0.
3
4  public class Point {
5      private int x;
6      private int y;
7
8      // Constructs a new Point at the given initial x/y position.
9      public Point(int initialX, int initialY) {
10         setLocation(initialX, initialY);
11     }
12
13     // Returns the direct distance between this Point and the origin (0, 0).
14     public double distanceFromOrigin() {
15         return Math.sqrt(x * x + y * y);
16     }
17
18     // Returns the x-coordinate of this Point.
19     public int getX() {
20         return x;
21     }
22
23     // Returns the y-coordinate of this Point.
24     public int getY() {
25         return y;
26     }
27
28     // Sets this Point's x/y coordinates to the given values.
29     public void setLocation(int newX, int newY) {
30         if (newX < 0 || newY < 0) {
31             throw new IllegalArgumentException();
32         }
33
34         x = newX;
35         y = newY;
36     }
37
38     // Shifts the position of this Point object by the given amount
39     // in each direction.
40     public void translate(int dx, int dy) {
41         setLocation(x + dx, y + dy);
42     }
43 }
```

Here's the corresponding fourth version of our client program. The client is actually a bit less clean now, because it must now call the getX and getY methods on a Point to print its coordinates. However, this moderate inconvenience to the client code is necessary to preserve the encapsulation and class invariant on our Point objects.

```
1  // A program that deals with 2D points.
2  // Fourth version, to accompany encapsulated Point class.
3
4  public class PointMain {
5      public static void main(String[] args) {
6          // create two Point objects
7          Point p1 = new Point(7, 2);
8          Point p2 = new Point(4, 3);
9
10         // print each point and its distance from origin
11         System.out.println("p1 is (" + p1.getX() + ", " + p1.getY() + ")");
12         System.out.println("distance from origin = " + p1.distanceFromOrigin());
13
14         System.out.println("p2 is (" + p2.getX() + ", " + p2.getY() + ")");
15         System.out.println("distance from origin = " + p2.distanceFromOrigin());
16
17         // translate each point to a new location
18         p1.translate(11, 6);
19         p2.translate(1, 7);
20
21         // print the points again
22         System.out.println("p1 is (" + p1.getX() + ", " + p1.getY() + ")");
23         System.out.println("p2 is (" + p2.getX() + ", " + p2.getY() + ")");
24     }
25 }
```

8.6 More Instance Methods

In this section we'll add some special instance methods to our Point to make it easier and more convenient to use in client code. We'll write a method to make it easy to print Point objects and another to compare them to each other for equality.

The toString Method

Java objects know how to print themselves because they contain a special method named toString. The toString instance method returns a String representation of the object. Java's Point type had a toString method that we used to write code like the following:

```
// using Java's Point type
Point p1 = new Point(7, 2);
System.out.println("p1 is " + p1.toString());
```

Recall that writing `.toString()` was optional in many cases, and the above println statement could have been written as:

```
System.out.println("p1 is " + p);
```

Either version of the previous code produces the output when run with Java's Point objects:

```
p is java.awt.Point[x=7,y=2]
```

Our own Point type doesn't have this functionality. When we try to run the same code with our own Point class, we get a strange result. The code does compiles and run, despite the fact that we haven't defined any toString method for our Point type. The output the toString() call produces on our Point objects is bizarre. Here's a possible output of the same code shown previously when run with our own Point class:

```
p = Point@119c082
```

The reason the code compiles, despite our not having written any toString method, is that there are certain methods every Java object contains by default. The toString method is one of these; two others are named equals and getClass. When defining a new type of objects, Java gives your objects a default version of each of these methods and a few others. If you like, you can replace the default version with one of your own. (This is actually an example of a concept called inheritance, which we'll discuss in the next chapter.)

In the case of toString, the default version of the method prints the type of the object, an @ sign, and a hexadecimal (base-16) number. This isn't a very useful message to represent a Point object, so we'll write our own toString method in the Point class that returns a String representing the state of a Point object. The toString method is an accessor that requires no parameters and has a return type of String.

```
// Returns a String representation of this Point object.
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

With our toString method, the same code sample shown previously now produces the following output. We didn't write our toString output to match that of Java's Point type, instead opting for a shorter and more readable version.

```
p = (7, 2)
```

Sun's Java guidelines recommend writing a toString method in every class you write.

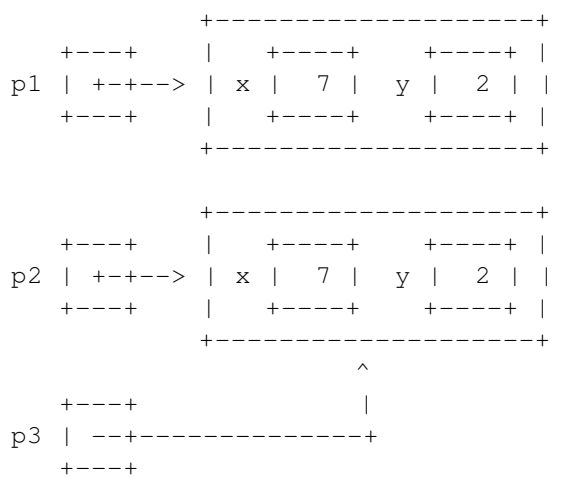
The equals Method

Every Java object contains a special method named equals that it uses to compare itself to other objects, returning true if the objects have the same state and false otherwise. We saw in previous chapters that the == operator does not behave properly when used on objects and that the equals method should be used instead for object comparisons.

Consider the following two Point objects referenced by three variables. The two objects are declared with the same state.

```
Point p1 = new Point(7, 2);   // using Java's Point type
Point p2 = new Point(7, 2);
Point p3 = p2;
```

The following diagram represents the state of these objects and the variables that refer to them. Notice that p3 is not a new object but a second reference to the same object as p2. This means that a change to p2, such as a call of its translate method, would also be reflected in p3.



In Chapter 3 we saw that the == operator performs a comparison of identity when used on objects. In other words, an expression of the form `a == b` evaluates to true if the two variables refer to the same object, and false otherwise. With the preceding Point objects, the expression `p1 == p2` would evaluate to false because p1 and p2 are not physically the same object. The expression `p2 == p3` would evaluate to true, because p3 refers to the same object as p2. This is not usually the behavior we want when comparing objects; often we want to know whether two different objects have the same internal state.

The equals method performs a comparison of two objects' states and returns true if the states are the same. With the preceding Point objects, the expressions `p1.equals(p2)`, `p1.equals(p3)`, and `p2.equals(p3)` all evaluate to true because the Points all have the same x/y coordinates.

If we used our own Point type to declare the same three Point variables, the equals method would not behave correctly. The code would compile successfully, but the equals method would behave the same way as the == operator: it would compare the references and only evaluate to true if the two variables referred to the same object. So the expression `p2.equals(p3)` would be true but the expression `p1.equals(p2)` would be false.

The reason the code compiles at all is because, like toString, equals is a method that every Java object contains, even if that object's class doesn't define an equals method. The default behavior of equals is to behave the same way as the == operator; that is, to compare the references of the two objects. As with the ugly default toString output, the default behavior often isn't what we want.

To correct the behavior of equals, we can define an equals method in our Point class. The equals method must accept an Object as its parameter and return a boolean value. A Point's equals method should test whether its own state is equal to that of the parameter object. For our Point objects, having the same state means having the same x and y data field values.

You might think that the following code correctly implements the equals method, but it has a flaw.

```
// A flawed implementation of an equals method.
public boolean equals(Point other) {
    return x == other.x && y == other.y;
}
```

The flaw in the preceding code is in the method's header. When you write an equals method, you're writing a method that compares this Point to any other object, not just other Point objects. For example, it should be legal to compare a Point and a String with an expression such as `p1.equals("hello");` the equals method should return false in such a case because the parameter isn't a Point.

The fix for the equals method header is to make it accept a parameter of type Object. Object is a class in Java that can represent any type of object. By saying that our method accepts an Object as a parameter, we're making it legal to pass any Java object as the parameter to the method. When the method is called, we'll need to examine the type of the object and return a false result if it isn't a Point. The following pseudocode shows the pattern to follow.

```
public boolean equals(Object o) {
    if (o is not a Point object) {
        return false.
    } else {
        compare the x and y values.
    }
}
```

The first thing our code needs to do is to exclude objects that aren't Points. There is an operator named instanceof that tests whether a variable refers to an object of a given type. The best way to examine the type of the Object o is to use instanceof to see whether o is an instance of the Point type. An instanceof test is a binary expression that takes the following form and evaluates to produce a boolean result:

```
<expression> instanceof <type>
```

The instanceof operator is strange because it isn't a method and therefore doesn't use parentheses or dots or any other notation. It is separated from its operands by spaces. The operand on the left side is generally the variable and the type on the right is the class you wish to test against.

In our case, the following code performs the instanceof test. Notice that we have to parenthesize the instanceof test because we wish to negate it; we're trying to exclude things that are not an instance of the Point type.

```
// reject non-Point objects
if (!(o instanceof Point)) {
    return false;
}
```

A nice side benefit of the instanceof operator is that it produces a false result when o is null. This will make sure that if the client code contains an expression such as `p1.equals(null)`, it will correctly return false rather than throwing a NullPointerException.

Now that we've checked for non-Point objects, we want to handle the case where Object o does refer to a Point object. You might think that the following code would correctly compare the two Point objects and return the proper result. Unfortunately it does not even compile successfully:

```
return x == o.x && y == o.y;   // This code does not compile
```

The preceding code produces errors such as the following for each of o's data fields we try to access:

```
Point.java:36: cannot find symbol
symbol  : variable x
location: class java.lang.Object
```

The Java compiler doesn't allow us to write an expression such as `o.x` because it doesn't know ahead of time whether o's object will have a data field named x. This is the case even though our code tests to make sure that o is of type Point; the compiler still doesn't trust us.

If we want to treat o as a Point object, we must cast it from type Object to type Point. You have used typecasting to convert between primitive types, such as casting double to int. Casting between object types has a different meaning. A cast of an object is a promise to the compiler. The cast is your assurance that the reference actually refers to a different type and that the compiler can treat it as that type. In our case, we'll write a statement that casts o into a Point object, so the compiler will trust that we can access its x and y data fields. Here's the correct code:

```
// cast into Point and compare
Point other = (Point) o;
return x == other.x && y == other.y;
```

As you can see, writing a correct equals method is tricky. We have to jump over several hurdles related to working with types of objects and placating the Java compiler. The benefit of all this is that now our Point objects can be compared for equality against any other object.

Putting it all together, here is the finished code for our equals method.

```
// Returns whether o refers to a Point with the same x/y
// coordinates as this Point.
public boolean equals(Object o) {
    if (!(o instanceof Point)) {
        return false;
    }

    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

This code also illustrates a subtlety about the private keyword. Notice that the body of equals refers to the other.x and other.y fields directly. This is actually legal; private data fields are visible to all objects of that class, so one Point can directly access the data of another.

Many classes implement equals methods like ours, so much of the preceding equals code can be reused as boilerplate code. Here's a template for a well-formed equals method. The instanceof test and typecast are likely the first two things you'll want to do in any equals method you write.

```
public boolean equals(Object o) {
    if ((!o instanceof <type>)) {
        return false;
    }

    <type> other = (<type>) o;
    <compare the data and return the result>;
}
```

Here is the fifth version of our Point class after all of the changes in this section, including the toString and equals methods.

```
 1  // A Point object represents an ordered pair of (x, y) coordinates.
 2  // Fifth version: Advanced features (toString, equals,
 3  // and a class invariant that x, y >= 0).
 4
 5  public class Point {
 6      private int x;
 7      private int y;
 8
 9      // Constructs a new Point at the given initial x/y position.
10      // pre: initialX >= 0 && initialY >= 0
11      public Point(int initialX, int initialY) {
12          setLocation(initialX, initialY);
13      }
14
15      // Returns the direct distance between this Point and the origin (0, 0).
16      public double distanceFromOrigin() {
17          return Math.sqrt(x * x + y * y);
18      }
19
20      // Returns the x-coordinate of this Point.
21      public int getX() {
22          return x;
23      }
24
25      // Returns the y-coordinate of this Point.
26      public int getY() {
27          return y;
28      }
29
30      // Returns whether o refers to a Point with the same x/y
31      // coordinates as this Point.
32      public boolean equals(Object o) {
33          if (!(o instanceof Point)) {
34              return false;
35          }
36
37          Point other = (Point) o;
38          return x == other.x && y == other.y;
39      }
40
41      // Sets this Point's x/y coordinates to the given values.
42      // pre: newX >= 0 && newY >= 0
43      public void setLocation(int newX, int newY) {
44          if (newX < 0 || newY < 0) {
45              throw new IllegalArgumentException();
46          }
47
```

```
48          x = newX;
49          y = newY;
50      }
51
52      // Returns a String representation of this Point object.
53      public String toString() {
54          return "(" + x + ", " + y + ")";
55      }
56
57      // Shifts the position of this Point object by the given amount
58      // in each direction.
59      public void translate(int dx, int dy) {
60          setLocation(x + dx, y + dy);
61      }
62  }
```

Here is the final version of the client program, which now takes advantage of the toString method when printing Point objects.

```
1   // A program that deals with 2D points.
2   // Fifth version, to accompany Point class with toString method.
3
4   public class PointMain {
5       public static void main(String[] args) {
6           // create two Point objects
7           Point p1 = new Point(7, 2);
8           Point p2 = new Point(4, 3);
9
10          // print each point and its distance from origin
11          System.out.println("p1 is " + p1);
12          System.out.println("distance from origin = " + p1.distanceFromOrigin());
13
14          System.out.println("p2 is " + p2);
15          System.out.println("distance from origin = " + p2.distanceFromOrigin());
16
17          // translate each point to a new location
18          p1.translate(11, 6);
19          p2.translate(1, 7);
20
21          // print the points again
22          System.out.println("p1 is " + p1);
23          System.out.println("p2 is " + p2);
24      }
25  }
```

8.7 The this Keyword

When we discussed instance methods, we saw that an instance method has an implicit knowledge of the object it's operating upon and can directly refer to that object's data fields and methods. But we didn't discuss the mechanics of exactly how this works in Java. The reason that instance methods know which object they're operating upon is because of a special reference named `this`. When you refer to a data field such as x in your code, really you're referring to `this.x`.

Anywhere in the Point class that you refer to a data field or method of the current Point, you may optionally write `this.` in front of the field or method name. For example, here's an implementation of the translate method that uses the `this` keyword when accessing its data fields. This style is less common, but some programmers prefer it because it is more explicit about what is going on.

```
// An implementation of translate using the this keyword.
public void translate(int dx, int dy) {
    this.x += dx;   // same as  x += dx;
    this.y += dy;   // same as  y += dy;
}
```

Thinking about the `this` keyword can present a clearer picture of the mechanics of what is really going on during an instance method call. Earlier in the chapter we diagrammed the behavior of some instance method calls on two Point objects. Let's revisit the same example with the `this` keyword. Consider the following two Point objects:

```
Point p1 = new Point(7, 2);
Point p2 = new Point(4, 3);
```

After constructing the Point objects we make the following method calls:

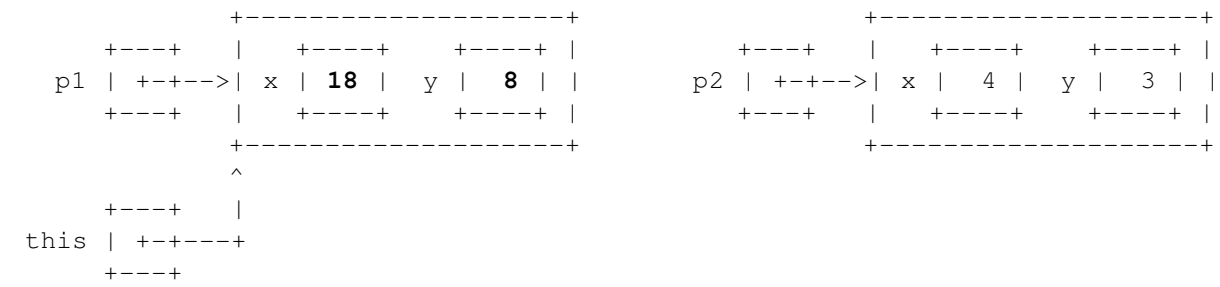
```
p1.translate(11, 6);
p2.translate(1, 7);
```

Essentially the behavior of these two instance method calls is the following:

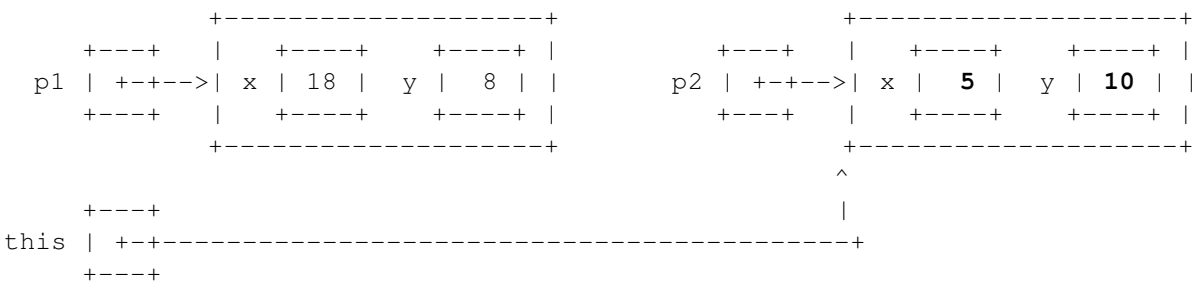
- Set Point class's `this` reference to refer to the same object as p1.
- Execute the translate method with parameters (11, 6).

- Set Point class's `this` reference to refer to the same object as p2.
- Execute the translate method with parameters (1, 7).

During the first instance method call, the `this` reference refers to the same object as p1. Therefore the method call adjusts the x/y coordinates of p1's object.



During the second method call, the `this` reference refers to the same object as p2. So the lines in the body of the translate method change the x and y fields of p2's object.



The `this` keyword does have some other uses. You can use `this` if you need the current object to pass itself as a parameter to a method. For example, if you want to print the current object's state for debugging after it's been constructed, you can write a println statement with `this` as the parameter.

```
public Point(int initialX, int initialY) {
    this.setLocation(initialX, initialY);
    System.out.println(this);   // for debugging
}
```

A common usage of the `this` keyword is to deal with shadowed variables. A *shadowed variable* occurs when two variables with the same name exist at the same place in the code. Normally this can't happen because we can't declare two variables with the same name. But one exception to this rule is that it is legal for an object's data field to have the same name as a parameter or local variable in one of its methods. So it's legal to have a method header such as the following, even though our data fields are named x and y:

```
public void setLocation(int x, int y) {
```

Shadowing occurs here because if you refer to x, the parameter x will be used and not the data field x. However, if you write `this.x`, the data field x will be used. Some programmers like this style where variables take the same name as fields if they will be stored into those fields. It saves you from having to use strange parameter names like initialX or newY. Here's an example of the setLocation method using x and y as its parameter names:

```
// Sets this Point's x/y coordinates to the given values.
public void setLocation(int x, int y) {
    if (x < 0 || y < 0) {
        throw new IllegalArgumentException();
    }

    this.x = x;
    this.y = y;
}
```

The principle of shadowing applies the same to constructors. We can modify our Point constructor's parameter names to be just x and y as well, since the constructor's body passes their values to setLocation.

```
// Constructs a new Point at the given initial x/y position.
public Point(int x, int y) {
    setLocation(x, y);
}
```

Multiple Constructors

The this keyword can also be used to help write multiple constructors. It's legal for a class to have more than one constructor as long as they accept different parameters. Presently our Point class has a constructor that requires two parameters: the Point's initial x and y coordinates.

```
// Constructs a new Point at the given initial x/y position.
public Point(int x, int y) {
    setLocation(x, y);
}
```

Before we wrote our constructor, it was legal to create a Point object that represented the origin, (0, 0), by passing no parameters when constructing it. This isn't legal any more, now that we added our two-parameter constructor. However, we can make it legal again by adding a second constructor to our Point class. Our new, parameterless constructor might look like this:

```
// Constructs a Point object at the origin, (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```

Now it would be possible to construct Points in two ways:

```
Point p1 = new Point(5, -2);   // (5, -2)
Point p2 = new Point();        // (0, 0)
```

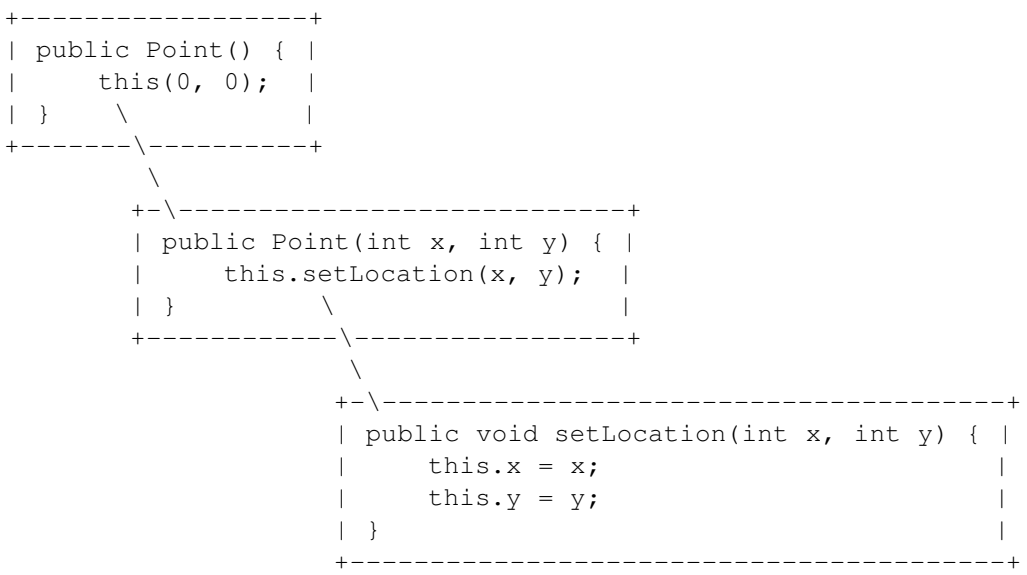
But by adding a second constructor, we have again introduced a bit of redundancy into our class. Both constructors perform similar actions; the only difference is exactly what initial value the x and y data fields receive. A useful observation here is that the new constructor can be expressed in terms of the old constructor. The following two lines have the same effect:

```
Point p1 = new Point();        // construct Point at (0, 0)
Point p2 = new Point(0, 0);    // construct Point at (0, 0)
```

We can express this in our Point class by having the parameterless constructor call the two-parameter constructor. The syntax for one constructor to call another is a bit strange: in its body we write the keyword `this`, followed by the parameters to pass to the other constructor in parentheses. In our case, we want to pass parameter values of 0 and 0 to initialize each field.

```
// Constructs a new Point object at the origin, (0, 0).
public Point() {
    this(0, 0);
}
```

The usage of `this` causes the parameterless constructor to utilize the behavior of the other constructor. The following diagram depicts this behavior. Notice that we're also using the `this` keyword to call methods and to disambiguate our data fields from our parameters.



The general syntax for one constructor to call another is the following:

```
this(<expression>, <expression>, ..., <expression>);
```

This is really just the normal syntax for a method call, except for the fact that we use the special `this` keyword where we would normally put the name of a method.

Here is the final version of our Point class after all of the changes in this section. This version of the class uses the `this` keyword throughout. The client program is unmodified because it does not construct any Points at (0, 0).

```
1  // A Point object represents an ordered pair of (x, y) coordinates.
2  // Sixth version: multiple constructors and this keyword.
3
4  public class Point {
5      private int x;
6      private int y;
7
8      // Constructs a new Point object at the origin, (0, 0).
9      public Point() {
10         this(0, 0);
11     }
12
13     // Constructs a new Point at the given initial x/y position.
14     public Point(int x, int y) {
15         setLocation(x, y);
16     }
17
18     // Returns the direct distance between this Point and the origin (0, 0).
19     public double distanceFromOrigin() {
20         return Math.sqrt(x * x + y * y);
21     }
22
23     // Returns the x-coordinate of this Point.
24     public int getX() {
25         return x;
26     }
27
28     // Returns the y-coordinate of this Point.
29     public int getY() {
30         return y;
31     }
32
33     // Returns whether o refers to a Point with the same x/y
34     // coordinates as this Point.
35     public boolean equals(Object o) {
36         if (!(o instanceof Point)) {
37             return false;
38         }
39
40         Point other = (Point) o;
41         return x == other.x && y == other.y;
42     }
43
44     // Sets this Point's x/y coordinates to the given values.
45     public void setLocation(int x, int y) {
46         if (x < 0 || y < 0) {
47             throw new IllegalArgumentException();
48         }
49
50         this.x = x;
51         this.y = y;
52     }
53
54     // Returns a String representation of this Point object.
55     public String toString() {
56         return "(" + x + ", " + y + ")";
57     }
58
59     // Shifts the position of this Point object by the given amount
```

```
60      // in each direction.
61      public void translate(int dx, int dy) {
62          setLocation(x + dx, y + dy);
63      }
64  }
```

8.8 Case Study: Designing a Stock Class

In this section, we'll create another new class of objects called Stock, for a program that manages a portfolio of shares of stock that the user has purchased. The user can add a new stock to his/her portfolio or can record purchases of shares of a stock at a certain price. This information can be used to report the investor's profit or loss on that stock. The interaction with the program might look like this:

```
First stock's symbol: AMZN
How many purchases did you make? 2
1: How many shares, at what price per share? 50 35.06
2: How many shares, at what price per share? 25 38.52
What is today's price per share? 37.29
Net profit/loss: $80.75

Second stock's symbol: INTC
How many purchases did you make? 1
1: How many shares, at what price per share? 15 16.50
What is today's price per share? 17.80
Net profit/loss: $19.5

AMZN was more profitable than INTC.
```

The stock program could be written using existing types such as doubles and Strings, but it would be cumbersome to keep track of all the variables, especially if we purchased multiple stocks, because each would need a copy of those variables.

To make the client program shorter and simpler, let's write a Stock class. Each Stock object would remember the shares and total price of a single stock and could report the profit or loss based on the current daily stock price.

Remember that when we define new types of objects, we start to have programs that occupy more than one file. The StockManager program itself will reside in the file StockManager.java, and the new Stock class we'll write will go into a separate file Stock.java. When we compile and run StockManager.java, it will also depend on Stock.java to run properly. We'll keep these two files in the same folder so that the Java compiler can find them and compile them together.

Stock Behavior

When designing a class, the first question to be asked is, "What behavior should these objects have?" We started our Point class by specifying its data fields, but that was because fields are conceptually simpler than instance methods. The objects' behavior is what the client program will use, so having the right behavior in each object is very important.

Another issue to consider is the division of behavior between the client program and the Stock class. Which code should go in what file? A good rule of thumb is to only store behavior in the Stock class if it's closely related to stock data. Another useful design rule is to avoid putting console I/O code (such as System.out.println statements or Scanner commands) in your Stock class. The I/O code is best left in the client program, so that the Stock class can be used with many different kinds of client programs.

A Stock object needs to be able to perform the following behavior:

- recording a purchase of shares of the stock at a given price
- reporting the total profit/loss on the stock, based on the current price per share of the stock

We'll represent these behaviors as methods named purchase and getProfit. The purchase method will accept parameters for the number of shares and price per share. The getProfit method will accept the current price per share of the stock and will return the total profit.

Here's a skeleton of our Stock class so far:

```
// Incomplete Stock class.
public class Stock {
    <fields>

    <constructor(s)>

    public double getProfit(double currentPrice) {
        <implementation>
    }

    public void purchase(int shares, double pricePerShare) {
        <implementation>
    }
}
```

Stock Fields

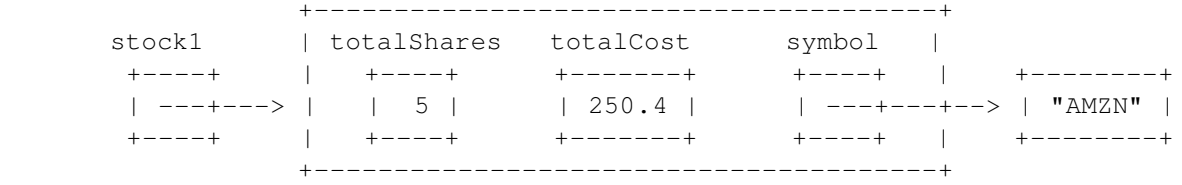
After deciding on the required behavior for a class, the next question to be asked is, "What data should each Stock object store to implement its behavior?" Each stock in the client program has several related pieces of data:

- the stock symbol (such as AMZN)
- the number of shares purchased
- the cost for the shares that have been purchased

We could try to store each purchase of the stock separately, but all we really need to know is the total number of shares in all purchases and the total cost of all shares combined. With this in mind, here is a list of the fields our Stock objects will need:

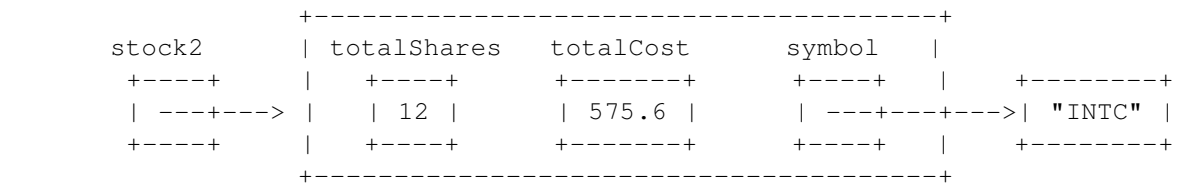
```
private String symbol;      // stock symbol, such as "YHOO"
private int totalShares;    // total number of shares purchased
private double totalCost;   // total cost for all shares purchased
```

A variable "stock1" referring to a Stock object with symbol of "AMZN" and 5 total shares purchased at a total cost of $250.40 would look like this:



(The symbol field has an arrow to an external box because symbol is a reference to a String object. This is an example of how objects' data fields can be references to other objects.)

Many Stock objects can be created, and each will have its own copy of the data fields. It would be possible to create a second Stock object with symbol "INTC" and 12 shares purchased at a total cost of $575.60. If it were stored in a variable named "stock2", its state would look like this:



The current price per share is also a piece of data related to a Stock. But we won't treat it as a data field because it's only used to find the Stock's profit. If a value is only used in one method of the class and is passed as a parameter to that method, it's best not to make it a data field. Cluttering a class with too many data fields reduces the coherence of the object and can make the code harder to read.

Stock Constructor

Now that we've decided on the Stock's state and behavior, let's think about how to construct Stock objects so that the client program can create two stocks and record purchases of them.

It may be tempting to write a constructor that accepts three parameters: one for each of the Stock's data fields. But the initial state of a given Stock object is that no shares have been purchased. So we actually know the initial total shares (0) and the initial total cost (0.00) without the client's help. Therefore our Stock constructor should accept just one parameter: the symbol for the Stock. We will save the symbol value into the object's `symbol` data field.

```
// Creates a new Stock with the given symbol and no shares purchased.
public Stock(String theSymbol) {
    symbol = theSymbol;
    totalShares = 0;
    totalCost = 0.00;
}
```

When a constructor takes an object as an parameter (such as the String symbol), it might make sense to check that parameter's value to make sure it isn't null. One possible way to handle this case would be to throw an exception if a null symbol is passed when creating a Stock object. The following lines could be inserted at the start of the Stock's constructor:

```
if (theSymbol == null) {
    throw new NullPointerException("null symbol");
}
```

Sun recommends that you throw a NullPointerException when a parameter's value is null but should not be. For other invalid parameter values, throw an IllegalArgumentException.

Here's a skeleton of our Stock class so far:

```
// Incomplete Stock class with fields and constructor.
public class Stock {
    private String symbol;      // stock symbol, such as "YHOO"
    private int totalShares;    // total number of shares purchased
    private double totalCost;   // total cost for all shares purchased

    // Creates a new Stock with the given symbol and no shares purchased.
    public Stock(String theSymbol) {
        if (theSymbol == null) {
            throw new NullPointerException("null symbol");
        }

        symbol = theSymbol;
        totalShares = 0;
        totalCost = 0.00;
    }

    public double getProfit(double currentPrice) {
        <implementation>
    }

    public void purchase(int shares, double pricePerShare) {
        <implementation>
    }
}
```

Stock Method Implementation

Now that we have created the Stock state and constructor, let's write the bodies of our methods. First we'll look at the purchase method, which accepts a number of shares and a price per share as parameters and modifies the Stock object's state to record that purchase. Recording the purchase consists of adding the number of shares to our total, and adding the price paid for these shares to our total cost. The price paid is equal to the number of shares times the price per share. Here's the code for the purchase method to implement this behavior:

```
// Records a purchase of the given number of shares of this stock
// at the given price per share.
public void purchase(int shares, double pricePerShare) {
    totalShares += shares;
    totalCost += shares * pricePerShare;
}
```

As with the constructor, it might make sense here to check the parameters passed in to make sure they are valid. In this case a valid number of shares and price per share would be ones that are non-negative. The following lines could be inserted at the start of our purchase method to perform this test:

```
if (shares < 0 || pricePerShare < 0) {
    throw new IllegalArgumentException("negative shares or price");
}
```

Now let's write the getProfit method, which computes how much money we've made or lost on our Stock. The profit of a Stock is equal to its market value minus what we paid for it: if we paid $20 for our shares and they now have a market value of $30, our profit is $10. We already know what we paid for the Stock, because we're keeping track of it in our totalCost field. But we don't know the market value yet, so we'll have to compute it by multiplying our totalShares field by the current price per share. Once we have the market value, we'll just subtract the totalCost we paid to find out our profit.

```
// Returns the total profit or loss earned on the shares of this stock,
// based on the given price per share.
public double getProfit(double currentPrice) {
    double marketValue = totalShares * currentPrice;
    return marketValue - totalCost;
}
```

As with the other methods, we should check for illegal parameter values. In this case we shouldn't allow a negative current price per share, so the following code can be placed at the start of the getProfit method:

```
if (currentPrice < 0.0) {
    throw new IllegalArgumentException("negative current price");
}
```

There are other methods that would be useful to have in our Stock objects and are excluded for space reasons. For example, it would probably also be good to implement accessors for the Stock's data: the symbol, number of shares, and so on. Also useful would be a toString method so that Stock objects could be printed easily. These methods are left for you to implement as exercises.

The Complete Stock Class

After all data fields, constructor, and methods of our Stock class are written, the class would look like this.

```
1  // A Stock object represents purchases of shares of a particular stock.
2  public class Stock {
3      private String symbol;        // stock symbol, such as "YHOO"
4      private int totalShares;      // total number of shares purchased
5      private double totalCost;     // total cost for all shares purchased
6
7      // Creates a new Stock with the given symbol and no shares purchased.
8      // Precondition: symbol != null
9      public Stock(String theSymbol) {
10         if (theSymbol == null) {
11             throw new NullPointerException("null symbol");
12         }
13
14         symbol = theSymbol;
15         totalShares = 0;
16         totalCost = 0.00;
17     }
18
19     // Returns the total profit or loss earned on the shares of this stock,
20     // based on the given price per share.
21     // Precondition: currentPrice >= 0.0
22     public double getProfit(double currentPrice) {
23         if (currentPrice < 0.0) {
24             throw new IllegalArgumentException("negative current price");
25         }
26
27         double marketValue = totalShares * currentPrice;
28         return marketValue - totalCost;
29     }
30
31     // Records a purchase of the given number of shares of this stock
32     // at the given price per share.
33     // Precondition: numShares >= 0 && pricePerShare >= 0.00
34     public void purchase(int shares, double pricePerShare) {
35         if (shares < 0 || pricePerShare < 0.0) {
36             throw new IllegalArgumentException("negative shares or price");
37         }
38
39         totalShares += shares;
40         totalCost += shares * pricePerShare;
41     }
42 }
```

The client code to use the Stock class might look like this.

```
1  // This program tracks the user's purchases of two stocks,
2  // computing and reporting which stock is more profitable.
3
4  public class StockManager {
5      public static void main(String[] args) {
6          Scanner console = new Scanner(System.in);
7
8          // first stock
9          System.out.print("First stock's symbol: ");
10         String symbol1 = console.next();
11
12         // Create a Stock object to represent my purchases of this stock
13         Stock stock1 = new Stock(symbol1);
14         double profit1 = makePurchases(stock1, console);
15
16         // second stock
17         System.out.print("Second stock's symbol: ");
18         String symbol2 = console.next();
19         Stock stock2 = new Stock(symbol2);
20         double profit2 = makePurchases(stock2, console);
21
22         // report which stock made more money
23         if (profit1 > profit2) {
24             System.out.println(symbol1 + " was more " +
25                     "profitable than " + symbol2 + ".");
26         } else if (profit2 > profit1) {
27             System.out.println(symbol2 + " was more " +
28                     "profitable than " + symbol1 + ".");
29         } else {
30             System.out.println(symbol1 + " and " + symbol2 +
31                     " are equally profitable.");
32         }
33     }
34
35     // Makes several purchases of stock and then returns the profit.
36     public static double makePurchases(Stock currentStock, Scanner console) {
37         System.out.print("How many purchases did you make? ");
38         int numPurchases = console.nextInt();
39
40         // Ask about each purchase
41         for (int i = 1; i <= numPurchases; i++) {
42             System.out.print(i + ": How many shares, at what price per share? ");
43             int numShares = console.nextInt();
44             double pricePerShare = console.nextDouble();
45
46             // Ask the Stock object to store this purchase
47             currentStock.purchase(numShares, pricePerShare);
48         }
49
50         // Use the Stock object to compute how much money I made / lost
51         System.out.print("What is today's price per share? ");
52         double currentPrice = console.nextDouble();
53
54         double profit = currentStock.getProfit(currentPrice);
55         System.out.println("Net profit/loss: $" + profit);
56         System.out.println();
57         return profit;
58     }
59 }
```

Chapter Summary

- Object-oriented programming is a different philosophy of writing programs that focuses on nouns or entities in a program, rather than verbs or actions of a program. In object-oriented programming, state and behavior are grouped into objects that communicate with each other.
- A class serves as the blueprint for a new type of objects, specifying their data and behavior. The class can be asked to construct many objects (also called "instances") of its type.
- The data for each object is specified using special variables called fields.
- The behavior of each object is specified by writing instance methods in the class. Instance methods exist inside an object and can access and act on that object's internal state.
- A class can define a special method named a constructor that creates and returns a new object and initializes its state. The constructor is the method that will be called when external client code creates a new object of your type using the `new` keyword.
- Usually one object can communicate with others despite not knowing all details about how the other objects work, a principle known as abstraction. Most objects protect their internal data from unwanted external modification, a principle known as encapsulation. Encapsulation is provided by declaring fields with the `private` modifier, which prevents other classes from directly modifying their values.
- Two common categories of object methods are accessors and mutators. Accessors provide us with information about the object, such as the length method of a String object or the getX method of a Point object. Mutators allow us to modify the state of the object, such as the translate method of a Point object.
- Objects can be made easily printable by writing a `toString` method. Objects can be made testable for equality by writing an `equals` method.
- The `this` keyword can be used when an object wishes to refer to itself. The `this` keyword is also used when a class has multiple constructor and one constructor wishes to call another.

Self-Check Problems

Section 8.1: Object-Oriented Programming Concepts

1. Describe the difference between object-oriented programming and procedural programming.
2. What is an object? How is an object different from a class?
3. Give three examples of types of objects you've used so far.

Section 8.2: Object State: Fields

4. Explain the differences between a data field and a parameter. What is the difference in their syntax? What is the difference in their scope and how they may be used?

Section 8.3: Object Behavior: Methods

5. Explain the differences between a static method and an instance method. What is the difference in their syntax? What is the difference in how they are used?

6. What is the difference between an accessor and a mutator? What are the naming conventions used with accessors and mutators?

Section 8.4: Object Initialization: Constructors

7. What is a constructor? How is a constructor different from a method?

8. What are two major problems with the following constructor?

```
public void Point(int initialX, int initialY) {
    int x = initialX;
    int y = initialY;
}
```

Section 8.5: Encapsulation

9. What is abstraction? How do objects provide an abstraction?

10. What is the difference between the `public` and `private` keywords? What items should be declared `private`?

11. When data fields are made `private`, client programs cannot see them directly. How do you allow classes access to read these fields' values, without letting the client break the object's encapsulation?

Section 8.6: More Instance Methods

12. How do you write a class whose objects can be easily printed on the console?

13. Write a modified version of the `toString` method on the Point class, such that its toString output matches the format of the toString from Java's Point objects. Here is an example:

```
java.awt.Point[x=3,y=-8]
```

Section 8.7: The this Keyword

14. What is the meaning of the `this` keyword? How is it used with data fields and methods?

15. Add a constructor to the Point class that accepts another Point as a parameter and initializes this new Point to have the same x/y values.

16. Why doesn't the == operator work correctly with objects? How do you specify a way to compare objects of your class for equality?

Section 8.8: Case Study: Designing a Stock Class

17. Add the following methods to the Stock class:

    `public String getSymbol()`
    Returns this Stock's symbol value, such as "AMZN".

    `public int getShares()`
    Returns this Stock's number of shares value.

    `public double getTotalCost()`
    Returns this Stock's total cost value.

Exercises

1. Add the following method to your Point class:

   `public double getDistance(Point other)`
   Returns the distance between the current Point object and the given other Point object. The distance between two points (*x1, y1*) and (*x2, y2*) can be expressed as the square root of $(x2 - x1)^2 + (y2 - y1)^2$. Two points with the same (x, y) coordinates should return a distance of 0.0.

2. Add the following method to your Point class:

   `public int getManhattanDistance(Point other)`
   Returns the "Manhattan distance" between the current Point object and the given other Point object. The Manhattan distance refers to how far apart two places are if the person can only travel straight horizontally or vertically, as though driving on the streets of Manhattan. In our case, the Manhattan distance is the sum of the absolute values of the differences in their coordinates; in other words, the difference in x plus the difference in y between the points.

3. Add the following method to your Point class:

`public double getSlope(Point other)`
Returns the slope of the line drawn between this Point and the given other Point. Use the formula (y2 - y1) / (x2 - x1) to determine the slope between two points (x1, y1) and (x2, y2). Note that this formula fails for points with identical x coordinates, so throw an IllegalArgumentException in this case.

4. Add the following method to your Point class:

   `public boolean isColinear(Point p1, Point p2)`
   Returns whether this Point is colinear with the given two other points. Points are colinear if a straight line can be drawn that connects them. Two basic examples are three points that have the same x or y coordinate. The more general case can be determined by calculating the slope of the line between each pair of points and seeing that this slope is the same for all pairs of points. Use the formula (y2 - y1) / (x2 - x1) to determine the slope between two points (x1, y1) and (x2, y2). (Note that this formula fails for points with identical x coordinates so this will have to be special-cased in your code.)

   Since Java's double type is imprecise, round all slope values to a reasonable accuracy such as 4 digits past the decimal point before you compare them.

5. Add the following static method to your Point class:

   `public static Point parsePoint(String str)`
   Examines the given String and converts it into an appropriate Point object, which is returned. For example, `Point.parsePoint("(-2, 3)")` should return a new Point with x value of -2 and y value of 3. It should always be the case for any Point p that `Point.parsePoint(p.toString()).equals(p)`.

   If you have rewritten your toString method to match that of Java's Point class, make your parsePoint method use that format. For example, `Point.parsePoint("java.awt.Point[x=-2, y=3]")` should return a new Point with x value of -2 and y value of 3.

6. Add the following method to the Stock class provided:

   `public void clear()`
   Resets this Stock's number of shares purchased and total cost to 0.

7. Add the following method to the Stock class provided:

   `public String toString()`
   Returns a String representation of this Stock object, such as `"AMZN (75 shares, $2716.0 total cost)"`.

8. Add the following method to the Stock class provided:

   `public boolean equals(Object o)`
   Returns true if the given object o is a Stock object with the same symbol, number of shares purchased, and total cost as this one.

9. Write a class named `Line` that represents a line segment between two Points. Your Line objects should have the following methods:

`public Line(Point p1, Point p2)`
Constructs a new Line that contains the given two Points.

`public Point getP1()`
Returns this Line's first endpoint.

`public Point getP2()`
Returns this Line's second endpoint.

`public String toString()`
Returns a String representation of this Line, such as "[(-2, 3), (4, 7)]".

10. Add the following method to your Line class:

    `public double getSlope()`
    Returns the slope of this Line. The slope of a line between points (x1, y1) and (x2, y2) is equal to (y2 - y1) / (x2 - x1). If x2 == x1, the denominator is zero and the slope is undefined, so you may throw an Exception in this case.

11. Add the following constructor to your Line class:

    `public Line(int x1, int y1, int x2, int y2)`
    Constructs a new Line that contains the given two Points.

12. Add the following method to your Line class:

    `public boolean equals(Object o)`
    Returns whether the given other object o is a Line with the same endpoints as this Line.

13. Add the following method to your Line class:

    `public boolean isColinear(Point p)`
    Returns true if the given point is colinear with the points of this Line; that is, if this Line stretched infinitely, would it hit the given Point? Points are colinear if a straight line can be drawn that connects them. Two basic examples are three points that have the same x or y coordinate. The more general case can be determined by calculating the slope of the line between each pair of points and seeing that this slope is the same for all pairs of points. Use the formula (y2 - y1) / (x2 - x1) to determine the slope between two points (x1, y1) and (x2, y2). (Note that this formula fails for points with identical x coordinates so this will have to be special-cased in your code.)

    Since Java's double type is imprecise, round all slope values to a reasonable accuracy such as 4 digits past the decimal point before you compare them.

14. Write a class named `Rectangle` that represents a rectangular 2-dimensional region. Your Rectangle objects should have the following methods:

    `public Rectangle(int x, int y, int width, int height)`
    Constructs a new Rectangle whose top-left corner is specified by the given coordinates and with the given width and height. Throw an IllegalArgumentException on a negative width or height.

`public int getHeight()`
Returns this Rectangle's height.

`public int getWidth()`
Returns this Rectangle's width.

`public int getX()`
Returns this Rectangle's x coordinate.

`public int getY()`
Returns this Rectangle's y coordinate.

`public String toString()`
Returns a String representation of this Rectangle, such as "Rectangle[x=1,y=2,width=3,height=4]".

15. Add the following method to your Rectangle class:

`public boolean equals(Object o)`
Returns whether the given other object o is a Rectangle with the same x/y coordinates, width, and height as this Rectangle.

16. Add the following constructor to your Rectangle class:

`public Rectangle(Point p, int width, int height)`
Constructs a new Rectangle whose top-left corner is specified by the given Point and with the given width and height.

17. Add the following methods to your Rectangle class:

```
public boolean contains(int x, int y)
public boolean contains(Point p)
```
Returns whether the given Point or coordinates lie inside the bounds of this Rectangle.

18. Add the following method to your Rectangle class:

`public Rectangle union(Rectangle rect)`
Returns a new Rectangle that represents the area occupied by the tightest bounding box that contains both this rectangle and the given other rectangle.

19. Add the following method to your Rectangle class:

`public Rectangle intersection(Rectangle rect)`
Returns a new Rectangle that represents the largest rectangular region completely contained within both this rectangle and the given other rectangle. If the rectangles do not intersect at all, returns a rectangle whose width and height are both 0.

Programming Projects

1. Write a class named `RationalNumber` that represents a fraction with an integer numerator and denominator. A RationalNumber object should have the following methods:

`public RationalNumber(int numerator, int denominator)`
Constructs a new rational number to represent the ratio (numerator/denominator). The denominator cannot be 0, so throw an IllegalArgumentException if 0 is passed.

`public RationalNumber()`
Constructs a new rational number to represent the ratio (0/1).

`public boolean equals(Object o)`
Returns whether the given object o is a RationalNumber object with equal value to this one. Two rational numbers need not have identical numerator and denominator to be considered equal; ratios such as (3/5) and (6/10) have equal value, so they too should be considered equal.

`public int getDenominator()`
Returns this rational number's denominator value; for example, if the ratio is 3/5, returns 5.

`public int getNumerator()`
Returns this rational number's numerator value; for example, if the ratio is 3/5, returns 3.

`public String toString()`
Returns a String representation of this rational number, such as "3/5".

2. Write a class named `Date` that represents a date consisting of a year, month, and day. A Date object should have the following methods:

`public Date(int year, int month, int day)`
Constructs a new Date to represent the given date.

`public void addDays(int days)`
Moves this Date object forward in time by the given number of days.

`public void addWeeks(int weeks)`
Moves this Date object forward in time by the given number of 7-day weeks.

`public int daysTo(Date other)`
Returns the number of days that this Date must be adjusted to make it equal to the given other Date.

`public boolean equals(Object o)`
Returns whether two Date objects represent the same calendar date.

`public int getDay()`
Returns this Date's day value; for example, for the date 2006/07/22, returns 22.

`public int getMonth()`
Returns this Date's month value; for example, for the date 2006/07/22, returns 7.

`public int getYear()`
Returns this Date's year value; for example, for the date 2006/07/22, returns 2006.

```
public String toString()
```
Returns a String representation of this Date in Year/Month/Day order, such as "2006/07/22".

3. Write a class named `GroceryList` that represents a person's list of items to buy from the market, and another class named `GroceryItemOrder` that represents a request to purchase a particular item in a given quantity (example: 4 boxes of cookies).

   The GroceryList class should use an array field to store the grocery items, as well as keeping track of its size (number of items in the list so far). Assume that a grocery list will have no more than 10 items. A GroceryList object should have the following methods:

   ```
   public GroceryList()
   ```
   Constructs a new empty grocery list.

   ```
   public void add(GroceryItemOrder item)
   ```
   Adds the given item order to this list, if the list is not full (has fewer than 10 items).

   ```
   public double getTotalCost()
   ```
   Returns the total sum cost of all grocery item orders in this list.

   The GroceryItemOrder class should store an item quantity and price per unit. A GroceryItemOrder object should have the following methods:

   ```
   public GroceryItemOrder(String name, int quantity, double pricePerUnit)
   ```
   Constructs an item order to purchase the item with the given name, in the given quantity, which costs the given price per unit.

   ```
   public double getCost()
   ```
   Returns the total cost of this item in its given quantity. For example, 4 boxes of cookies that are 2.30 per unit have a cost of 9.20.

   ```
   public void setQuantity(int quantity)
   ```
   Sets this grocery item's quantity to be the given value.

Stuart Reges
Marty Stepp

Chapter 9
Inheritance and Interfaces

Copyright © 2006 by Stuart Reges and Marty Stepp

- 9.1 Inheritance Concepts
  - Non-programming Hierarchies
- 9.2 Programming with Inheritance
  - Overriding Methods
  - Polymorphism
- 9.3 The Mechanics of Polymorphism
  - Diagramming Polymorphic Code
- 9.4 Interacting with the Superclass
  - Inherited Fields
  - Calling a Superclass's Constructor
  - Calling Overridden Methods
  - A Larger Example: Point3D
- 9.5 Inheritance in the Java Class Libraries
  - Graphics2D (optional)
  - Input/Output Streams
- 9.6 Interfaces
  - An Interface for Shape Classes
  - Implementing the Shape Interface
  - Benefits of Interfaces
  - Interfaces in the Java Class Libraries
- 9.7 Case Study: Designing a Hierarchy of Financial Classes
  - Class Design
  - Initial Redundant Implementation
  - Abstract Classes

Introduction

In this chapter we will explore two of the most important techniques provided by the Java language to write better structured solutions. Inheritance allows us to share code between classes to reduce redundancy, as well as letting us treat different types of objects in the same way. Interfaces allow us to treat several different types of objects the same way without sharing code.

9.1 Inheritance Concepts

As we begin to learn about inheritance, we'll first look at the genesis of how inheritance came about, as well as some non-programming examples of the ideas behind inheritance. This will lead us toward programming with inheritance in Java.

Did You Know: The Software Crisis

Software has been getting more and more complicated since the advent of programming. Around the beginning of the 1970s, it was becoming clear that some common problems existed when teams wrote such larger and more complex programs. Projects were running over budget; they were not finishing on time; the software had bugs, didn't do what it was supposed to do, or was otherwise of low quality. Collectively, these problems came to be called the "software crisis."

A particularly sticky issue was with program maintenance. Companies found that they spent much of their time not writing new programs but modifying and maintaining existing ones. This proved to be a difficult task, because it was easy to write disorganized and redundant code. Changes performed during maintenance work were likely to take a long time and to introduce new bugs into the system.

The negative effects of the software crisis and maintenance programming were particularly noticeable when graphical user interfaces came into prominence in the 1980s. User interfaces in graphical systems like Microsoft Windows and Apple's Macintosh were much more sophisticated than the text interfaces that preceded them. The original graphical programs were prone to redundancy because they had to describe in detail how buttons, text boxes, and other onscreen components were implemented. Also the graphical components themselves contain a lot of common state and behavior, such as a size, shape, color, position, or scrollbar.

Large programs demand the ability to write versatile and clear code on a large scale. In this textbook so far, we've seen several ways to express programs more concisely and elegantly on a small scale. Features like static methods, parameterization, loops, and classes help us organize our programs and extract common features that can be used in many places. This general practice is called *code reuse*.

Code Reuse

The practice of writing program code once and using it in many contexts.

Object-oriented programming provides us with a feature called inheritance that enables us to reuse code, including making entire classes reusable in different contexts. Inheritance also gives us the benefit of writing programs with hierarchies of related object types.

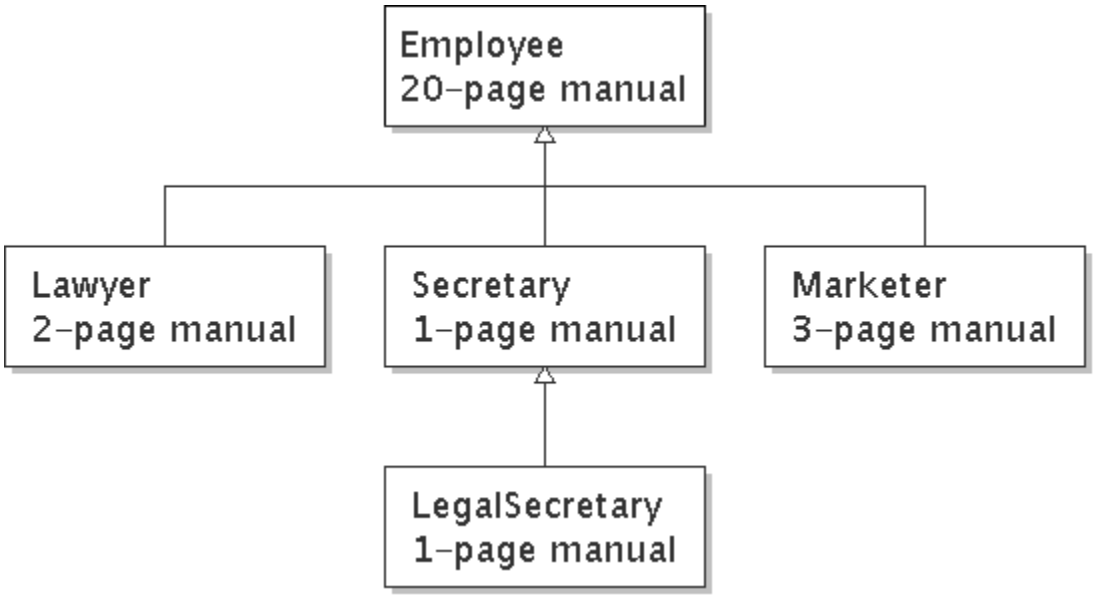
Non-programming Hierarchies

When we start to think about inheritance, we'll want to be able to identify similarities between different objects and classes in our program. Let's start with a non-programming example: A hierarchy of employees at a company.

Imagine a large law firm that hires several types of employees: lawyers, general secretaries, legal secretaries, and marketers. The company has a bunch of employee rules about vacation and sick days, medical benefits, harassment regulations, and so on. Each subdivision of the company also has a few of its own rules; for example, maybe lawyers ask for vacation leave with a different form than secretaries do.

Imagine that all the employees attend a common orientation whele they learn the general rules. The employee is given a 20-page manual of these rules to read. A mixed group of employees could attend the orientation together: lawyers, secretaries and marketers all might sit in the same orientation group.

Afterward, the employee goes to his/her subdivision and receives a second, smaller orientation covering any rules specific to that division. The employee receives a smaller manual, 2 or 3 pages in length, covering that subdivision's specific rules. Some rules are added to those in the general 20-page manual, and a few are replaced. For example, perhaps secretaries get 3 weeks of vacation instead of 2; perhaps lawyers use a pink form to apply for time off, not the yellow form listed in the 20-page manual. Each subdivision has its own submanual, and their lengths and contents may differ.



An alternative solution would be to give every employee a large manual containing both the applicable general rules and the rules of their subdivision. There would be a 22-page manual for the lawyers, a 21-page manual for secretaries, and a 23-page manual for marketers. In fact, the consolidation might even save a few pages. So why does the company bother with two manuals for every employee?

The main issues are with redundancy and maintenance. The 22-page lawyer manual contains a lot of the same text as the 21-page secretary manual. If a common rule is changed, all the manuals need to be updated individually, which is a tedious and error-prone process. There's also a certain visual appeal to the pair of manuals. If someone wants to know all rules that are specific to lawyers, they can simply read the 2-page lawyer manual, rather than combing through a 22-page lawyer manual trying to spot differences.

There are two key ideas here:

1. It's useful to be able to specify a broad set of rules that will apply to many related groups (the 20-page manual).
2. It's also useful for a particular group to be able to specify a smaller set of rules specific to itself, including the ability to replace some rules from the broad set ("Use the pink form instead of the yellow form").

An important thing to notice about the categories is that they nest within each other. For example, every legal secretary is also a secretary and every marketer is also an employee. In a pinch, you could ask a legal secretary to work as a standard secretary for a short period, because a legal secretary is a secretary. We call such a connection an *is-a relationship*.

Is-a Relationship

A hierarchical connection between two categories where one type can be treated as (possibly a specialized version of) the other.

When applying these concepts to programming, each type of employee might be represented as a class. The different employee groups represent a set of related classes connected by is-a relationships. We call such a set of classes an *inheritance hierarchy*.

Inheritance Hierarchy

A set of hierarchical relationships between types of objects.

As we'll see, inheritance hierarchies are commonly used in Java to group related types of objects and reuse code between them.

9.2 Programming with Inheritance

The previous section was a non-programming example of hierarchies. But as an exercise, we could write small Java classes to represent those categories of employees as classes. The code will be a bit silly but will illustrate some important concepts.

Let's imagine that we have the following rules for our employees:

- All employees work 40 hours per week.
- All employees make $40,000 salary per year, except marketers who make $50,000 per year.
- All employees have 2 weeks of paid vacation leave per year, except lawyers who have 3 weeks of vacation leave.
- All employees use a yellow form to apply for vacation leave, except lawyers who use a special pink form.
- Each employee type has one unique ability: lawyers know how to sue, marketers know how to advertise, and secretaries know how to take dictation.

Let's write a class to represent the common behavior of all employees. (Think of this as the 20-page employee manual.) We'll write methods named showHours, showSalary, showVacation and applyForVacation to represent these behaviors. To keep things simple, each method will just print a short relevant String representing the default employee behavior, such as $40,000 salary and the yellow form for vacation leave. We won't declare any data fields for now.

```
1  // A class to represent employees in general.
2  public class Employee {
3      public void applyForVacation() {
4          System.out.println("Use the yellow form to apply for vacation.");
5      }
6
7      public void showHours() {
8          System.out.println("I work 40 hours per week.");
9      }
10
11     public void showSalary() {
12         System.out.println("My salary is $40,000.");
13     }
14
15     public void showVacation() {
16         System.out.println("I receive 2 weeks of paid vacation.");
17     }
18 }
```

Now let's think about implementing the Secretary subcategory. One thing we mentioned in the previous section was that every Secretary is also an Employee, and they should retain the abilities that Employees have. If we wrote Secretary in a straightforward way, its code probably would not reflect this relationship very elegantly. The Secretary class would be forced to redundantly repeat all of the same methods from Employee with identical behavior. The only change to Secretary would be to add a takeDictation method to represent the unique behavior of a Secretary. Here is the redundant class:

```
1  // A redundant class to represent secretaries.
2  public class Secretary {
3      public void applyForVacation() {
4          System.out.println("Use the yellow form to apply for vacation.");
5      }
6
7      public void showHours() {
8          System.out.println("I work 40 hours per week.");
9      }
10
11     public void showSalary() {
12         System.out.println("My salary is $40,000.");
13     }
14
15     public void showVacation() {
16         System.out.println("I receive 2 weeks of paid vacation.");
17     }
18
19     // This is the only unique behavior.
20     public void takeDictation() {
21         System.out.println("I know how to take dictation.");
22     }
23 }
```

What we'd really like to be able to do is say the following:

```
// This is pseudocode.
public class Secretary {
    <copy all the methods from the Employee class.>

    // This is the only unique behavior.
    public void takeDictation() {
        System.out.println("I know how to take dictation.");
    }
}
```

Fortunately, Java provides a mechanism called *inheritance* that can help us remove this sort of redundancy between similar classes of objects. Inheritance allows the programmer to specify a relationship between two classes where one class absorbs ("inherits") the state and behavior of another.

Inheritance (inherit)

A programming technique in which a derived class extends the functionality of a base class, inheriting all of its state and behavior.

The child class, more commonly called the *subclass*, inherits all of the state and behavior of its parent type, commonly called the *superclass*. The child can then add existing state and behavior of its own, or it can replace its inherited behavior with new behavior as needed.

Superclass

The parent class in an inheritance relationship.

Subclass

The child class in an inheritance relationship.

A Java class can have only one superclass; it is not possible to extend more than one class. This is called *single inheritance*. One class may have many subclasses extending it.

The general syntax to specify one class as the subclass of another is the following:

```
public class <subclass name> extends <superclass name> {
    ...
}
```

We could rewrite the Secretary class to extend the Employee class. This will create an is-a relationship where every Secretary also is an Employee. This means that Secretary objects will inherit copies of the applyForVacation, showHours, showSalary, and showVacation methods. We won't need to write these methods in the Secretary class any more. This will remove the redundancy because Secretaries perform these action the same way as general Employees.

It's legal and expected for a subclass to add new behavior that wasn't present in the superclass. We said previously that employees add an ability not seen in other persons: the ability to work. We can add this to our previously empty Employee class. Therefore, the only code we need to write in the Employee class is the work method. The following is the complete Secretary class:

```
1  // A class to represent secretaries.
2  public class Secretary extends Employee {
3      public void takeDictation() {
4          System.out.println("I know how to take dictation.");
5      }
6  }
```

Inheritance creates an is-a relationship, because each object of the subclass can be treated as an object of the superclass.

The following client code would work with our new Secretary class:

```
1  public class EmployeeMain {
2      public static void main(String[] args) {
3          System.out.println("Employee:");
4          Employee employee1 = new Employee();
5          employee1.applyForVacation();
6          employee1.showHours();
7          employee1.showSalary();
8          employee1.showVacation();
9          System.out.println();
10
11         System.out.println("Secretary:");
12         Secretary employee2 = new Secretary();
13         employee2.applyForVacation();
14         employee2.showHours();
15         employee2.showSalary();
16         employee2.showVacation();
17         employee2.takeDictation();
18     }
19 }
```

The code would produce the following output. Notice that the first four methods produce the same output for both objects, because Secretary inherits that behavior from Employee.

```
Employee:
Use the yellow form to apply for vacation.
I work 40 hours per week.
My salary is $40,000.
I receive 2 weeks of paid vacation.

Secretary:
Use the yellow form to apply for vacation.
I work 40 hours per week.
My salary is $40,000.
I receive 2 weeks of paid vacation.
I know how to take dictation.
```

Overriding Methods

We can use inheritance in our other types of Employees, making Lawyer and Marketer classes that are subclasses of Employee. But while the Secretary only added behavior to the standard Employee behavior, the Lawyer and Marketer need to replace some of the standard Employee behavior with their own. Lawyers receive 3 weeks of vacation and use a pink form to apply for vacation. Marketers receive $50,000 salary.

It's legal for us to replace superclass behavior by writing a new version of the relevant method(s) in our subclass. The new version will replace the one inherited from Employee. This idea of replacing behavior from the superclass is called *overriding*.

Override

To implement a new version of a method inherited from a superclass, replacing the superclass's version.

Overriding requires no special syntax. To override a method, just write the method you want to replace in the subclass. Its name and signature must exactly match that of the method from the superclass.

Here are the Lawyer and Marketer classes that extends Employee and override the relevant methods.

```
1  // A class to represent lawyers.
2  public class Lawyer extends Employee {
3      public void applyForVacation() {
4          System.out.println("Use the pink form to apply for vacation.");
5      }
6
7      public void showVacation() {
8          System.out.println("I receive 3 weeks of paid vacation.");
9      }
10 }
```

```
1  // A class to represent marketers.
2  public class Marketer extends Employee {
3      public void showSalary() {
4          System.out.println("My salary is $50,000.");
5      }
6  }
```

The following client program uses our Lawyer and Marketer classes.

```
1  public class EmployeeMain2 {
2      public static void main(String[] args) {
3          System.out.println("Lawyer:");
4          Lawyer employee1 = new Lawyer();
5          employee1.applyForVacation();
6          employee1.showHours();
7          employee1.showSalary();
8          employee1.showVacation();
9          System.out.println();
10
11         System.out.println("Marketer:");
12         Marketer employee2 = new Marketer();
13         employee2.applyForVacation();
14         employee2.showHours();
15         employee2.showSalary();
16         employee2.showVacation();
17     }
18 }
```

The program produces the following output.

```
Lawyer:
Use the pink form to apply for vacation.
I work 40 hours per week.
My salary is $40,000.
I receive 3 weeks of paid vacation.

Marketer:
Use the yellow form to apply for vacation.
I work 40 hours per week.
My salary is $50,000.
I receive 2 weeks of paid vacation.
```

Be careful not to confuse overriding with overloading. Overloading, introduced in Chapter 3, is where one class contains multiple methods with the same name (but with different parameters). Overriding is when a subclass rewrites a new version of its superclass's method, with exactly the same name and the same parameters.

Polymorphism

One very interesting and odd thing about inherited types is that it's legal for a variable of a superclass type to refer to an object of one of its subclass types. For example, the following is a legal assignment statement:

```
Employee employee1 = new Lawyer();
```

You've already seen cases where one type can store a value of another type: with primitive values. For example, an int value can be stored into a double variable. In those cases, the int value was automatically converted into a double when it was assigned.

When a subclass object is stored into a superclass variable, no such conversion occurs. The object referred to by employee1 really is a Lawyer object, not an Employee object. If we call methods on it, it will behave like a Lawyer object. For example, the call:

```
employee1.applyForVacation();
```

produces the following output:

```
Use the pink form to apply for vacation.
```

It turns out that this ability for variables to refer to subclass objects is one of the most crucially important ideas in object-oriented programming. It allows us to write flexible code that can interact with many types of objects in the same way. For example, we can write a method that accepts an Employee as a parameter, or returns an Employee, or create an array of Employee objects. In any of these cases, we can substitute an Secretary, Lawyer, or other subclass object of Employee and the code will still work. Even more importantly, the code will actually behave differently depending on which type of object is used, because each subclass overrides and changes some of the behavior from the superclass. This important ability for the same code to be used with several different types of objects is called *polymorphism*.

Polymorphism

The ability for the same code to be used with several different types of objects and behave differently depending on the actual type of object used.

Here is an example test file that uses Employee objects polymorphically as parameters to a static method.

```
 1  public class EmployeeMain3 {
 2      public static void main(String[] args) {
 3          Employee empl = new Employee();
 4          Lawyer law = new Lawyer();
 5          Marketer mark = new Marketer();
 6          Secretary sec = new Secretary();
 7
 8          printInfo(empl);
 9          printInfo(law);
10          printInfo(mark);
11          printInfo(sec);
12      }
13
14      public static void printInfo(Employee employee) {
15          employee.applyForVacation();
16          employee.showHours();
17          employee.showSalary();
18          employee.showVacation();
19          System.out.println();
20      }
21  }
```

Notice that the static method lets us pass many different types of employees as parameters, and it produces different depending on which type was passed. Polymorphism gives us this flexibility. The program produces the following output:

```
Use the yellow form to apply for vacation.
I work 40 hours per week.
My salary is $40,000.
I receive 2 weeks of paid vacation.

Use the pink form to apply for vacation.
I work 40 hours per week.
My salary is $40,000.
I receive 3 weeks of paid vacation.

Use the yellow form to apply for vacation.
I work 40 hours per week.
My salary is $50,000.
I receive 2 weeks of paid vacation.

Use the yellow form to apply for vacation.
I work 40 hours per week.
My salary is $40,000.
I receive 2 weeks of paid vacation.
```

9.3 The Mechanics of Polymorphism

Inheritance and polymorphism introduce some complex new mechanics and behavior into our programs. One useful way to get the hang of these mechanics is to perform some exercises to interpret the behavior of inheritance programs. The main goal of these exercises is to understand in detail what happens when a Java program with inheritance executes.

The EmployeeMain3 program in the last section serves as a template for a more general set of questions we might ask about inheritance hierarchies. The questions take this form: Given the following hierarchy of classes, what behavior would result if we created several objects of the different types and called various methods on them?

To achieve some polymorphism in these types of questions, we'll store the objects being examined into an array. In the case of our Employee hierarchy, it's legal for an Lawyer, Secretary, or any other subclass of Employee to reside as an element of an Employee[]. The following program produces the same output as the EmployeeMain3 from last section.

```
 1  public class EmployeeMain4 {
 2      public static void main(String[] args) {
 3          Employee[] employees = {new Employee(), new Lawyer(),
 4                  new Marketer(), new Secretary()};
 5
 6          // print information about each employee
 7          for (int i = 0; i < employees.length; i++) {
 8              employees[i].applyForVacation();
 9              employees[i].showHours();
10              employees[i].showSalary();
11              employees[i].showVacation();
12              System.out.println();
13          }
14      }
15  }
```

Even if you didn't understand inheritance, you might be able to deduce some things about the hierarchy from the classes' names and the relationships between employees in the real world. So let's take this sort of exercise one step further. We'll make the names of the classes into nonsensical phrases that bear no resemblance to real-world entities. Admittedly this is not something you'd do when really programming, but our goal here is to examine the mechanics of inheritance and polymorphism on their own.

Assume that the following classes have been defined:

```
public class A {
    public void method1() {
        System.out.println("A 1");
    }

    public void method2() {
        System.out.println("A 2");
    }

    public String toString() {
        return "A";
    }
}
```

```
public class B extends A {
    public void method2() {
        System.out.println("B 2");
    }
}
```

```
public class C extends A {
    public void method1() {
        System.out.println("C 1");
    }

    public String toString() {
        return "C";
    }
}
```

```
public class D extends C {
    public void method2() {
        System.out.println("D 2");
    }
}
```

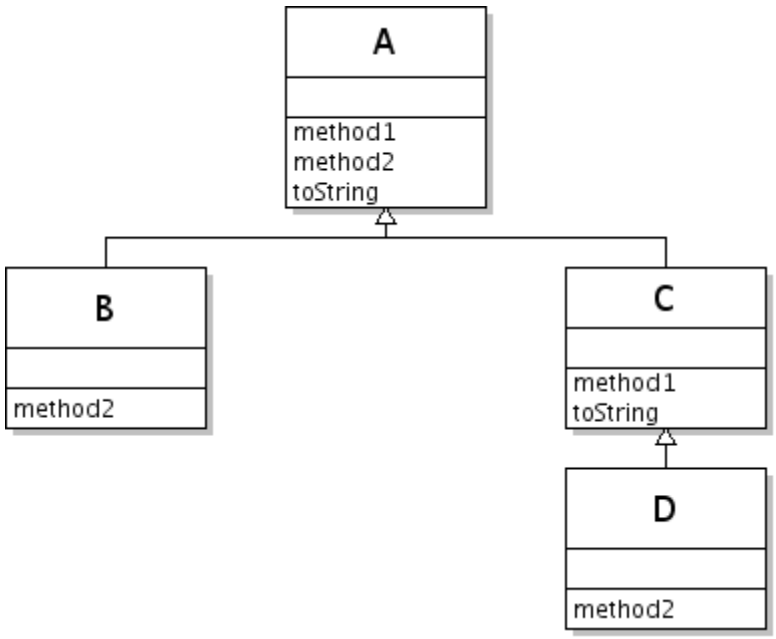
Consider the following client code, which uses the above classes. We declare an array of variables of a superclass type and fill it with objects of the various subclass types. When we call methods on the elements of our array, we should observe polymorphic behavior.

```
1  // Client program to use the A, B, C, and D classes.
2  public class ABCDMain {
3      public static void main(String[] args) {
4          A[] elements = {new A(), new B(), new C(), new D()};
5
6          for (int i = 0; i < elements.length; i++) {
7              System.out.println(elements[i]);
8              elements[i].method1();
9              elements[i].method2();
10             System.out.println();
11         }
12     }
13 }
```

How do we go about interpreting such code and determining its correct output?

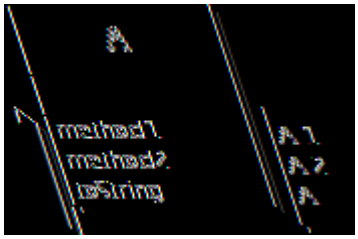
Diagramming Polymorphic Code

To determine the output of a polymorphic program like the one in the previous section, we must determine what happens when each element is printed (when its toString method is called), and when its method1 and method2 methods are called. One way to get started is to draw a diagram like those seen in previous sections. We'll draw each type as a box, listing its methods in the box, and connecting subclasses to their superclasses by arrows. (The pictures here are simplified versions of commonly used OOP drawings called UML Class Diagrams.)

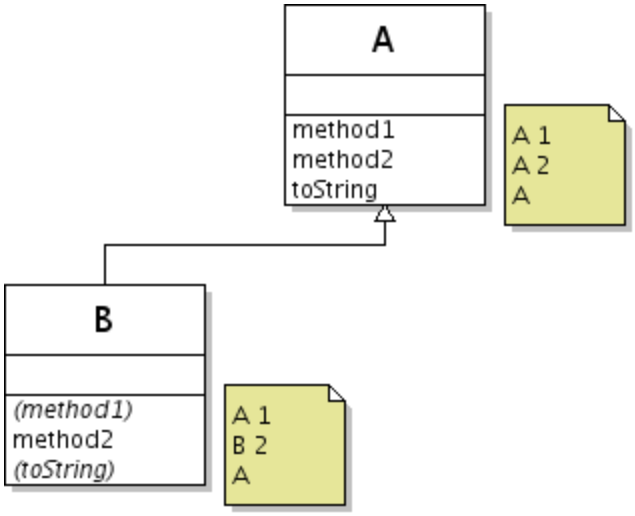


Now, let's enhance our diagram by writing the methods' output next to their names. Also, let's not only write the methods defined in each class, but also the ones that the class inherits.

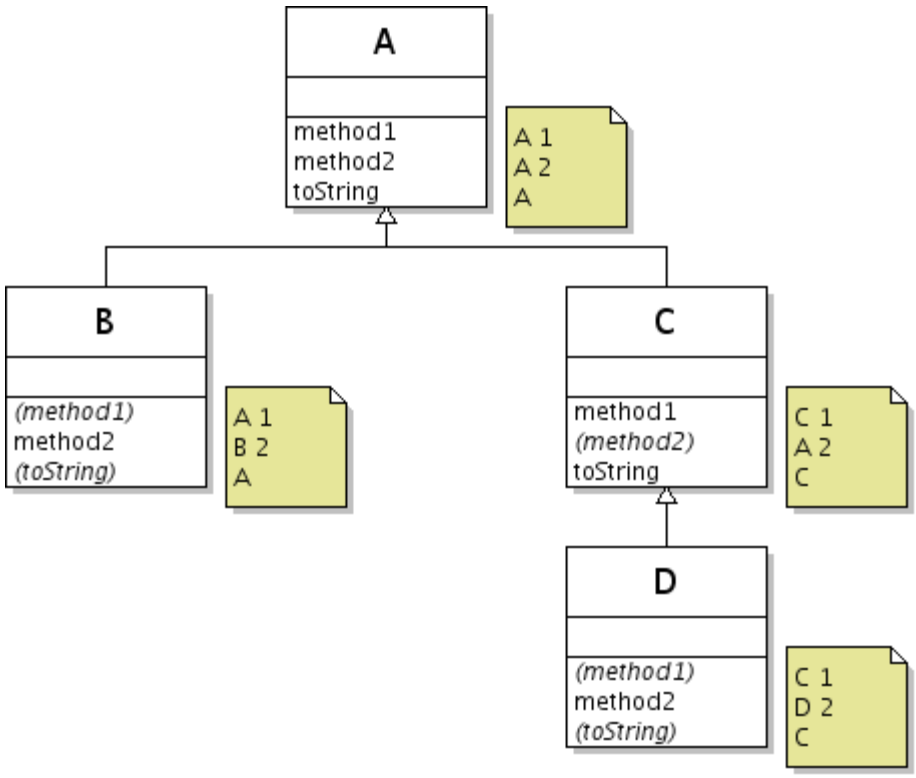
We'll start by filling in the methods from the A class with their output. When someone calls method1 on an A object, the resulting output is "A 1". When someone calls method2 on an A object, the resulting output is "A 2". When someone prints an A object with toString, the resulting output is "A". Let's fill them in as a note on our diagram:



Next let's look at the B type. B inherits all the behavior from A, except that it overrides the method2 output to say "B 2". So that means we can fill in the B output on our diagram identically to the A column except that we'll replace "A 2" with "B 2".



Third, let's look at the C type. C inherits all the behavior from A, except that it overrides the method1 output to say "C 1" and overrides the toString method to say "C". So that means we can fill in the C column of our table identically to the A column except that we'll replace "A 1" with "C 1" and "A" with "C". We'll treat the D type similarly. D inherits all the behavior from C, except that it overrides the method2 output to say "D 2". Following our pattern, we'll get this final table:



Another way to do this is to make a table of behavior for each type, for each of its methods. List each type horizontally and each method name vertically:

method name	**A**	**B**	**C**	**D**
method1	A 1	A 1	C 1	C 1
method2	A 2	B 2	A 2	D 2
toString	A	A	C	C

Now that we've created a diagram or table, we can figure out the output of the code. The array contains a A object, a B object, a C object, then a D object. On each of these it prints the toString output, then calls method1, then method2, then a blank line. When a method gets called on an object, we can just look up the output of that method for that type on our diagram or table. We can see that the output for these methods on an A object (element 0 of the array) is the following:

```
A
A 1
A 2
```

Using the table in this way for all 4 types, we get the following complete output for the exercise:

```
A
A 1
A 2

A
A 1
B 2

C
C 1
A 2

C
C 1
D 2
```

9.4 Interacting with the Superclass

The classes in the previous sections demonstrated inheritance with methods, but now we'll want to write more meaningful classes that use inheritance with fields, methods, and constructors. These subclasses require more complex interaction with the state and behavior they inherit from their superclass. To perform this interaction properly, we'll need to examine a new keyword named `super`.

Inherited Fields

In the previous chapter's case study, we built a Stock class representing purchased shares of a given stock. Here's the code for that class, shortened a bit for this section, and with a getTotalShares method added to return the stock owner's number of shares:

```
 1  // A Stock object represents purchases of shares of a particular stock.
 2  public class Stock {
 3      private String symbol;        // stock symbol, such as "YHOO"
 4      private int totalShares;      // total number of shares purchased
 5      private double totalCost;     // total cost for all shares purchased
 6
 7      // Creates a new Stock with the given symbol and no shares purchased.
 8      public Stock(String theSymbol) {
 9          symbol = theSymbol;
10          totalShares = 0;
11          totalCost = 0.00;
12      }
13
14      // Returns the total number of shares purchased of this stock.
15      public int getTotalShares() {
16          return totalShares;
17      }
18
19      // Returns the total profit or loss earned on this stock.
20      public double getProfit(double currentPrice) {
21          double marketValue = totalShares * currentPrice;
22          return marketValue - totalCost;
23      }
24
25      // Records a purchase of the given shares at the given price.
26      public void purchase(int shares, double pricePerShare) {
27          totalShares += shares;
28          totalCost += shares * pricePerShare;
29      }
30  }
```

Now let's imagine that we want to create a type of objects for stocks that pay dividends. Dividends are profit-sharing payments made by a corporation to its shareholders. The amount the shareholder receives is proportional to the number of shares they own. Not every stock pays dividends, so we wouldn't want to add this functionality directly to our Stock class. Instead, we'll create a new class called DividendStock that extends Stock and adds this new behavior.

Each DividendStock object will inherit the symbol, total shares, and total cost from the Stock superclass. We'll add a data field to record the amount of dividends paid.

```
public class DividendStock extends Stock {
    private double dividends;   // amount of dividends paid

    ...
}
```

Using the dividends data field, we can write a method in the DividendStock that lets the shareholder receive a per-share dividend. But the following code won't compile:

```
// This code does not compile!
public void payDividend(double amountPerShare) {
    dividends += amountPerShare * totalShares;
}
```

Common Programming Error: Trying to access a private data field from a subclass

A DividendStock cannot access the totalShares data field it has inherited, because totalShares is declared private in Stock. A subclass may not refer directly to any private data fields that were declared in its superclass. You'll get a compiler error like the following:

```
DividendStock.java:17: totalShares has private access in Stock
```

The solution is to use the accessor or mutator methods associated with those data fields to access or change their values. Here is a corrected version of the payDividend method:

```
// Records a payment of a divident of the given dividend
// amount for each share purchased of this stock.
public void payDividend(double amountPerShare) {
    dividends += amountPerShare * getTotalShares();
}
```

In a subclass, access or modify the superclass's data field using a method instead of referring to it directly.

Calling a Superclass's Constructor

Constructors (unlike methods) are not inherited, so we'll have to write our own constructor for the DividendStock class. The issue with accessing private data fields will arise again when we write a constructor for DividendStock.

We'd like the DividendStock constructor to accept the same parameter as the Stock constructor: the symbol. We want it to have the same behavior as the Stock constructor, but additionally we want to set the dividends field to 0.00. The following constructor implementation might seem like a good start, but it is redundant with Stock's constructor and also won't compile successfully:

```
// This constructor does not compile.
public DividendStock(String theSymbol) {
    symbol = theSymbol;
    totalShares = 0;
    totalCost = 0.00;
    dividends = 0.00;  // this line is the new code
}
```

The compiler produces four errors: one error for each line that tries to access an inherited private data field, and a message about a missing Stock() constructor.

```
DividendStock.java:5: cannot find symbol
symbol  : constructor Stock()
location: class Stock
    public DividendStock(String theSymbol) {
                                           ^
DividendStock.java:6: symbol has private access in Stock
DividendStock.java:7: totalShares has private access in Stock
DividendStock.java:8: totalCost has private access in Stock
```

The first problem is that even though a DividendStock does contain the symbol, totalShares, and totalCost data fields by inheritance, it cannot refer to them directly because they were declared private in the Stock class. It's done this way so that inheritance doesn't provide a loophole for a malicious subclass to break an object's encapsulation.

The second problem about a missing Stock() constructor is a subtle and confusing detail of inheritance. The problem is that a subclass's constructor must call a constructor from the superclass. Essentially the DividendStock partially consists of a Stock object, and the state of that Stock object must be initialized by calling a constructor for it. If we don't specify how to do so, the compiler assumes that Stock has a parameterless Stock() constructor and tries to initialize the Stock data by calling this constructor. Since the Stock class doesn't actually have a parameterless constructor, the bizarre error message about a missing constructor Stock() occurs. (It's a shame the error message isn't more informative.)

The solution to this problem is to explicitly call the Stock constructor that accepts a String symbol as its parameter. Java uses the keyword `super` for a subclass to refer to behavior from its superclass. To call a constructor of a superclass, write the keyword `super`, followed by the constructor's parameter values in parentheses.

```
super(<expression>, <expression>, ..., <expression>);
```

In the case of our DividendStock constructor, the following code does the trick. We'll use the `super` keyword to call the superclass constructor, passing it the same theSymbol value that was passed to our DividendStock constructor. This will initialize the symbol, total shares, and total cost. Then we set the initial dividends to 0.00 ourselves.

```
// Constructs a new dividend stock with the given symbol
// and no shares purchased.
public DividendStock(String theSymbol) {
    super(theSymbol);          // call Stock constructor
    this.dividends = 0.00;
}
```

The call to the superclass's constructor using `super` must be the first statement in a subclass's constructor. If you reverse the order of the statements in DividendStock's constructor and set the dividends before calling `super`, you'll get a compiler error like the following:

```
DividendStock.java:7: call to super must be first statement in constructor
        super(theSymbol);          // call Stock constructor
             ^
```

Here's our DividendStock class so far. The class isn't complete yet because we have not implemented the behavior to make dividend payments.

```
// Represents a stock purchase that also pays dividends.
public class DividendStock extends Stock {
    private double dividends;  // amount of dividends paid

    // Constructs a new dividend stock with the given symbol
    // and no shares purchased.
    public DividendStock(String theSymbol) {
        super(theSymbol);          // call Stock constructor
        this.dividends = 0.00;
    }
```

Although the `super` keyword helped us solve the issue of initializing our DividendStock, it's still a good idea to write accessor methods for the other fields in the superclass, such as getSymbol and getTotalCost, in case the DividendStock subclass or other classes need to use their values later.

Calling Overridden Methods

To implement dividend payments, we'll begin by writing a method named payDividend that accepts a dividend amount per share and adds the proper amount to our DividendStock's dividends data field. The amount per share is multiplied by the number of shares held.

```
// Records a payment of a divident of the given dividend
// amount for each share purchased of this stock.
public void payDividend(double amountPerShare) {
    dividends += amountPerShare * getTotalShares();
}
```

The dividend payments being recorded should be considered profit for the stock holder. The overall profit of a DividendStock is equal to the profit of the regular Stock plus the dividends. This is computed as the market value (number of shares times current price) minus the total cost paid for the shares, plus the amount of dividends paid.

Because the profit of a DividendStock is computed differently than that of a regular Stock, we should override the getProfit method in the DividendStock class to implement this new behavior. An incorrect initial attempt might look like this:

```
// This code does not compile.
public double getProfit(double currentPrice) {
    double marketValue = totalShares * currentPrice;
    return marketValue - totalCost + dividends;
}
```

The preceding code has two problems. For one, we cannot refer directly to the various data fields that were declared in Stock. We could add accessor methods for each data field to get around this. The second problem is that the code is redundant: it duplicates much of the functionality from Stock's getProfit method shown earlier. The only new behavior is the adding of dividends into the total.

To remove this redundancy, we can have DividendStock's getProfit method call Stock's getProfit method as part of its computation. However, since the two methods share the same name, we must disambiguate them by explicitly telling the compiler we wish to call Stock's version of the getProfit method. We do this using `super` keyword, which we also used in the previous section to call a superclass's constructor.

The general syntax for calling an overridden method using the super keyword is:

```
super.<method name>(<expression>, <expression>, ..., <expression>)
```

Here is the corrected code, which does compile and eliminates the previous redundancy:

```
// Returns the total profit or loss earned on this stock,
// including profits made from dividends.
public double getProfit(double currentPrice) {
    return super.getProfit(currentPrice) + dividends;
}
```

Here is the completed DividendStock class built in the preceding sections:

```
 1  // Represents a stock purchase that also pays dividends.
 2  public class DividendStock extends Stock {
 3      private double dividends;  // amount of dividends paid
 4
 5      // Constructs a new dividend stock with the given symbol
 6      // and no shares purchased.
 7      public DividendStock(String theSymbol) {
 8          super(theSymbol);          // call Stock constructor
 9          this.dividends = 0.00;
10      }
11
12      // Returns the total profit or loss earned on this stock,
13      // including profits made from dividends.
14      public double getProfit(double currentPrice) {
15          return super.getProfit(currentPrice) + dividends;
16      }
17
18      // Records a payment of a divident of the given dividend
19      // amount for each share purchased of this stock.
20      public void payDividend(double amountPerShare) {
21          dividends += amountPerShare * getTotalShares();
22      }
23  }
```

It's possible to have a deeper inheritance hierarchy with multiple layers of inheritance and overriding. However, the `super` keyword only reaches one level upward to the most recently overwritten version of the method. It's not legal to write super more than once in a row; you cannot write calls like `super.super.getProfit`. If you need such a solution, you'll have to find a workaround such as using different method names.

A Larger Example: Point3D

Let's consider another example of extending a class where `super` is useful. Imagine that we'd like to write a program that deals with points in 3-dimensional space, such as a 3D game, rendering program, or terrain simulation. A Point3D class would be useful for writing such programs. In Chapter 8 we already wrote a Point class to represent a point in 2D space. We can write our Point3D class to extend Point and add a z-coordinate.

Point3D's constructor needs to set the x and y fields, just as the Point constructor did. But Point3D adds a z field, which must also be set. Calling `super(x, y)` sets the first two fields; the third is set separately. Here's an initial version of the Point3D class with a z data field, constructor, and accessor for the z value:

```
1  // A Point3D object represents an ordered trio of coordinates (x, y, z).
2  public class Point3D extends Point {
3      private int z;
4
5      // Constructs a new Point3D object with the given coordinates.
6      public Point3D(int x, int y, int z) {
7          super(x, y);
8          this.z = z;
9      }
10
11     // Returns the z-coordinate of this Point3D.
12     public int getZ() {
13         return z;
14     }
15 }
```

One piece of useful behavior to add to our Point3D objects is a new toString method that includes the z coordinate as part of the String. We can override the toString method to implement this behavior. We'll have to call getX and getY to access our private x and y data fields when building the String.

```
// Returns a String representation of this Point3D.
public String toString() {
    return "(" + getX() + ", " + getY() + ", " + z + ")";
}
```

Another useful piece of behavior is a translate method that accepts dx, dy, and dz as parameters and shifts the point's 3D position. In this case, we'd like to call the superclass's version of translate to adjust the x- and y-coordinates, then adjust the z-coordinate ourselves. But our new translate method hasn't really overridden the one from Point. The translate method is not being overridden but overloaded: Point3D's code will be a second version of the same method but with different parameters.

The consequence of this is that when calling the superclass's version of translate, we wouldn't need to use the super keyword to disambiguate the two methods. The compiler can tell which method we're calling by looking at how many parameters we pass.

```
// Shifts the position of this Point3D object by the given amount.
public void translate(int dx, int dy, int dz) {
    translate(dx, dy);
    z += dz;
}
```

You do need to use the super keyword to call methods that you have overridden in your subclass, but you do not need it to call methods from the superclass that have not been overridden.

The following is the complete code for our Point3D class.

```
 1  // A Point3D object represents an ordered trio of coordinates (x, y, z).
 2  public class Point3D extends Point {
 3      private int z;
 4
 5      // Constructs a new Point3D object with the given coordinates.
 6      public Point3D(int x, int y, int z) {
 7          super(x, y);
 8          this.z = z;
 9      }
10
11      // Returns the z-coordinate of this Point3D.
12      public int getZ() {
13          return z;
14      }
15
16      // Returns a String representation of this Point3D.
17      public String toString() {
18          return "(" + getX() + ", " + getY() + ", " + z + ")";
19      }
20
21      // Shifts the position of this Point3D object by the given amount.
22      public void translate(int dx, int dy, int dz) {
23          translate(dx, dy);
24          z += dz;
25      }
26  }
```

9.5 Inheritance in the Java Class Libraries

Inheritance isn't only used in your code; it's also prevalent in the Java class libraries. In this section we'll look at two important uses of inheritance in the class libraries for drawing 2D graphics and performing input and output.

Graphics2D (optional)

The Java class libraries use inheritance in the drawing of 2D graphics. In Chapter 3's supplement on Graphics, we saw an object named Graphics that acts like a pen to draw shapes and lines onto a window. When Java's designers wanted additional graphical functionality, they extended the Graphics class into a more powerful class named Graphics2D. This is a good example of one of the more common uses of inheritance, to extend and reuse functionality from a powerful existing object.

Why didn't Sun simply add these new methods into the existing Graphics? There are several reasons. For one, it's preferable not to disturb a working piece of code if you can avoid it. Graphics already worked properly, so it was best not to perform unnecessary surgery on it. Second, making Graphics2D extend Graphics provides *backward compatibility*. Backward compatibility is the ability for new code to work correctly with old code without modification. By leaving Graphics untouched, old programs are guaranteed to remain working properly, while new programs can choose to use the new Graphics2D functionality if they wish.

Sun's documentation for Graphics2D describes the purpose of the class: "This Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. This is the fundamental class for rendering 2-dimensional shapes, text and images on the Java(tm) platform." To be specific, Graphics2D adds the ability to perform transformations such as scaling and rotation when you're drawing. These can lead to some fun and interesting images on the screen.

If you use the DrawingPanel class from Chapter 3's graphical supplement, you previously wrote statements like the following to get access to the panel's Graphics object:

```
Graphics g = panel.getGraphics();
```

It turns out that the getGraphics method really doesn't return a Graphics object at all, but rather a Graphics2D. Because of polymorphism, it was legal for your program to treat it as a Graphics, because every Graphics2D "is-a" Graphics. To use it as a Graphics2D object instead, simply say:

```
Graphics2D g2 = panel.getGraphics();
```

Here's a partial list of some of Graphics2D's extra methods:

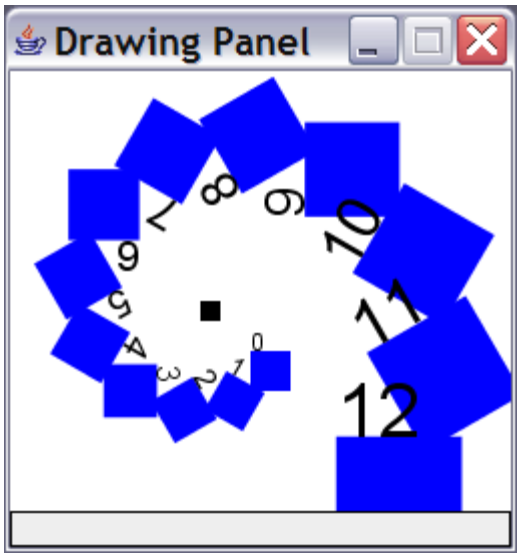
Useful Methods of Graphics2D Objects

`public void rotate(double angle)` Rotates subsequently drawn items by the given angle in radians with respect to the origin.
`public void scale(double sx, double sy)` Adjusts the size of any subsequently drawn items by the given factors (1.0 means equal size).
`public void shear(double shx, double shy)` Gives a slant to any subsequently drawn items.
`public void translate(double dx, double dy)` Shifts the origin by (dx, dy) amount in the current coordinate system.

The following program demonstrates Graphics2D. The rotate method's parameter is an angle of rotation measured in radians. A radian is an angular unit of measure equal to the value of π divided by 180. The Math class has a static method named toRadians that converts a number of degrees (in our program's case, 30) into the equivalent number of radians.

```
1  // Draws a picture of rotating squares using Graphics2D.
2  import java.awt.*;
3
4  public class FancyPicture {
5      public static void main(String[] args) {
6          DrawingPanel panel = new DrawingPanel(250, 220);
7          Graphics2D g2 = panel.getGraphics();
8          g2.translate(100, 120);
9          g2.fillRect(-5, -5, 10, 10);
10
11         for (int i = 0; i <= 12; i++) {
12             g2.setColor(Color.BLUE);
13             g2.fillRect(20, 20, 20, 20);
14
15             g2.setColor(Color.BLACK);
16             g2.drawString("" + i, 20, 20);
17
18             g2.rotate(Math.toRadians(30));
19             g2.scale(1.1, 1.1);
20         }
21     }
22 }
```

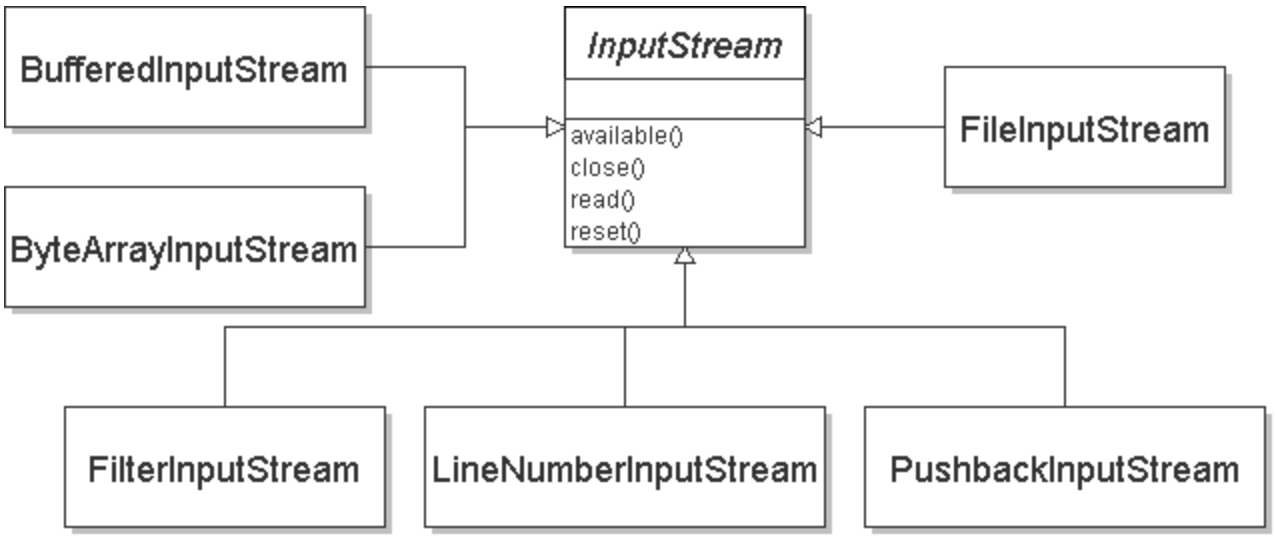
The program draws the following figure as its output:



Input/Output Streams

In previous chapters we've seen how to read and write data to the console, as well as reading and writing from external files. There are other potentially interesting sources of data that you might wish to use in your programs, such as from connections to another computer or a connection to a web page.

Java has a hierarchy of objects called *streams* that represent one-way flows of data. A stream could represent the bytes from a file, from the console, or elsewhere. The stream classes are found in the `java.io` package. When it comes to reading data, there's an overall superclass named `InputStream` and many subclasses representing the different possible sources of data. Part of the hierarchy looks like this:



It turns out that `System.in` is a variable in the `System` class, of type `InputStream`. When we construct a `Scanner` to read from the console, we're actually invoking a constructor that accepts an `InputStream` as a parameter.

```
public Scanner(InputStream in)
```

The other constructor we saw that accepts a `File` object actually uses that file to create a `FileInputStream` under the hood. It reads the file's data from this `FileInputStream`.

One interesting source of data we haven't yet explored is web pages. It turns out that it's not very difficult to open a `Scanner` to read data from a web page, but we will need to learn briefly about some new objects from the Java class libraries: `URL` and `IOException`.

A `URL` object from the `java.net` package represents a Uniform Resource Locator for a web page, such as http://www.apple.com/itunes/. We can use `URL` objects to represent a connection to a web page in our programs. To use `URL`, we'll need to import the `java.net` package into our program.

```
import java.net.*;   // for URL
```

The simplest way to construct a `URL` object is by passing a `String` as the parameter.

```
URL website = new URL("http://www.google.com/");
```

Once we create a `URL` object, we can ask it for an input stream from which to read the characters of that web page. A `URL` object has a method called `openStream` that returns an `InputStream`.

```
InputStream stream = website.openStream();
```

Now that we have the `InputStream` to read the web site, we can construct a `Scanner` to read the data. The `Scanner`'s usual methods such as `nextLine` and `hasNext` still work as we expect.

```
Scanner in = new Scanner(stream);
```

Unfortunately, the act of constructing a URL object and opening its input stream can potentially throw exceptions, such as if the URL String is badly formatted or if the web site / connection isn't available. The type of these exceptions is called IOException. (Constructing a URL object actually throws a subclass of IOException called MalformedURLException. Exceptions in Java use inheritance, too!) A program trying to use URLs would receive errors like these:

```
DisplayWebPage.java:7: unreported exception java.net.MalformedURLException;
must be caught or declared to be thrown
        URL website = new URL("http://www.google.com/");
                      ^
DisplayWebPage.java:8: unreported exception java.io.IOException;
must be caught or declared to be thrown
        InputStream stream = website.openStream();
                                                ^
2 errors
```

So we can either try/catch the exceptions to handle them, or we must declare our main method with throws IOException in its header. The following program reads and prints a web page. The program is declared to throw the exception, so if the web site or connection is down, the program will terminate with an exception error message.

```
 1  // This program reads the HTML text of a web page
 2  // and prints it on the console.
 3
 4  import java.io.*;
 5  import java.net.*;   // for URL
 6  import java.util.*;
 7
 8  public class DisplayWebPage {
 9      public static void main(String[] args) throws IOException {
10          URL website = new URL("http://www.homestarrunner.com/");
11          InputStream stream = website.openStream();
12          Scanner in = new Scanner(stream);
13          while (in.hasNextLine()) {
14              String line = in.nextLine();
15              System.out.println(line);
16          }
17      }
18  }
```

The program produces output such as the following:

```
<HTML>
<HEAD>
<TITLE>Everybody! Everybody!</TITLE>

</HEAD>
<BODY bgcolor="black">
<center><OBJECT classid="clsid:D27CDB6E-
AE6D-11cf-96B8-444553540000"

 codebase="http://active.macromedia.com/flash2/cabs/swflash.
 cab#version=4,0,0,0"
 ID=newintro WIDTH=550 HEIGHT=400>
 <PARAM NAME=movie VALUE="welcome.swf"> <PARAM NAME=quality
 VALUE=high> <PARAM NAME=bgcolor VALUE=#000000> <EMBED
 src="welcome.swf" quality=high  WIDTH=550 HEIGHT=400
 TYPE="application/x-shockwave-flash"
 PLUGINSPAGE="http://www.macromedia.com/shockwave/download/i
 ndex.cgi?P1_Prod_Version=ShockwaveFlash"></EMBED>
</OBJECT></center>
...
</BODY>
</HTML>
```

9.6 Interfaces

Inheritance is a very useful tool because it provides polymorphism and code-sharing, but it does have several limitations. For one, because Java uses single inheritance, a class can only extend one superclass. This makes it impossible to use inheritance to set up multiple is-a connections for types that wish to share multiple characteristics, such as objects that are both colored and comparable. There are also situations where you want polymorphism without sharing code, in which case inheritance isn't the right tool for the job.

Java has a construct called an *interface* that can represent a common supertype between several classes, without code sharing. An interface is like a class, but it only contains method headers without method bodies. An interface is like a set of qualifications that can be imposed on a class, or more specifically, a set of methods that a class can promise to implement.

Interface

A set of methods that classes can promise to implement, allowing those classes to be treated similarly in your code.

We say that an interface defines a *role* that an object can play; for example, a Rectangle class might implement the Comparable interface to indicate that Rectangle objects can be compared to each other, or a Point class might implement the Cloneable interface to indicate that a Point object can be replicated.

An Interface for Shape Classes

In this section we'll use an interface to define a polymorphic hierarchy of shape classes without sharing code between them. Imagine that we are creating classes to represent many different types of shapes, such as rectangles, circles, and triangles. We might be tempted to use inheritance with these shape classes because they seem to share some common behavior. All shapes have an area and perimeter, for example.

But each shape computes its area and perimeter in a totally different way, as listed below. The w and h represent the rectangle's width and height; the r represents the circle's radius; and the a, b, and c represent the lengths of the triangle's three sides. We'll defer the formula for a triangle's area for a moment.

shape	**area**	**perimeter**
Rectangle	w * h	2 (w + h)
Circle	πr^2	$2 \pi r$
Triangle	*see text*	a + b + c

It still seems as though there is an is-a relationship here, because a rectangle, circle, or triangle is a shape. But code-sharing isn't useful in this case because they all implement their behavior in completely different ways.

There's another more subtle problem with having a Shape superclass: it introduces a new unwanted type into the system. We wouldn't want a client program to be able to construct a `new Shape()` because there's no real-world entity called Shape. Every shape object should be one of the types we named or some other polygon.

A better solution would be to write an interface to represent the common functionality of all shapes: the ability to ask for their area and perimeter. The various shape classes can specify that they implement all methods of the Shape interface. We say that the classes "implement the interface," because they will implement the methods specified by the interface.

In our interface, we write a header To write an interface, we create a new .java file with the same name as the interface's name (our Shape interface would be stored in Shape.java). We give the interface a header with the keyword `interface` in place of the word `class`:

```
public interface Shape {
    <contents of the interface>
}
```

Inside the interface we write headers for each method we want shapes to contain. But instead of writing method bodies with { }, we simply place a semicolon at the end of the header. We don't specify how the methods are implemented. Instead we're demanding that any class that wants to be considered a Shape must implement these methods. (In fact, it isn't legal for an interface to contain method bodies; an interface can only contain method headers and class constants.)

The following is the complete code for our Shape interface. It declares that shapes have methods to compute their area and perimeter as type double.

```
1  // A general interface for shape classes.
2  public interface Shape {
3      public double getArea();
4      public double getPerimeter();
5  }
```

The methods of an interface are sometimes called *abstract methods* because we are only declaring their names and signatures, not specifying how they will be implemented.

Abstract Method

A method that is declared (as in an interface) but not implemented. Abstract methods represent the behavior a class promises to implement when it implements an interface.

Writing the `public` keyword on an interface's method headers is optional. We choose to include the `public` keyword so that the declarations in the interface match the headers of the method implementations in the classes. The general syntax we'll use for declaring an interface is the following:

```
public interface <name> {
    public <type> <name>(<type> <name>, ..., <type> <name>);
    public <type> <name>(<type> <name>, ..., <type> <name>);
    ...
    public <type> <name>(<type> <name>, ..., <type> <name>);
}
```

Declaring an interface doesn't actually define a new class of objects; if the client code tried to create a `new Shape()`, it would not compile. An interface only enumerates constraints to be placed on other types.

Implementing the Shape Interface

Now that we've written a Shape interface, we want to connect the various classes of shapes to it. If we want to connect a class to our interface with an is-a relationship, there are two things we have to do:

1. Declare that the class "implements" the interface
2. Implement each of the interface's methods in the class

The general syntax for declaring that a class implements an interface in a class is the following:

```
public class <name> implements <interface> {
    ...
}
```

In our various shape classes, we modify their headers to indicate that they implement all of the methods in the Shape interface. The file Rectangle.java, for example, might begin like this:

```
public class Rectangle implements Shape {
    ...
}
```

When we claim that our Rectangle implements Shape, we are promising that the Rectangle class will contain implementations of the getArea and getPerimeter methods. If a class claims to be a Shape but does not have a suitable getArea or getPerimeter method, it will not compile. For example, if we leave the body of our Rectangle class empty and try to compile it, the compiler would give errors like the following:

```
Rectangle.java:2: Rectangle is not abstract and does not override abstract method getPerime
public class Rectangle implements Shape {
       ^
1 error
```

The solution is to implement the getArea and getPerimeter methods in our Rectangle class. Let's define a Rectangle object by a width and height. Since the area of a rectangle is equal to its width times its height, we'll implement the getArea method by multiplying its data fields. We'll use the preceding perimeter formula 2 * (w + h) to implement getPerimeter. Here is the complete Rectangle class that implements the Shape interface:

```
 1  // Represents rectangle shapes.
 2  public class Rectangle implements Shape {
 3      private double width;
 4      private double height;
 5
 6      // Constructs a new rectangle with the given dimensions.
 7      public Rectangle(double width, double height) {
 8          this.width = width;
 9          this.height = height;
10      }
11
12      // Returns the area of this rectangle.
13      public double getArea() {
14          return width * height;
15      }
16
17      // Returns the perimeter of this rectangle.
18      public double getPerimeter() {
19          return 2.0 * (width + height);
20      }
21  }
```

The other classes of shapes are implemented in a similar fashion. We'll define a Circle object to have a field named radius. Notice that we don't have any common code between Circle and Rectangle, so inheritance is unnecessary. Here is the complete Circle class:

```
// Represents circle shapes.
public class Circle implements Shape {
    private double radius;

    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }

    // Returns the area of this circle.
    public double getArea() {
        return Math.PI * radius * radius;
    }

    // Returns the perimeter of this circle.
    public double getPerimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```

As for implementing the Triangle, we'll specify the shape by its three side lengths *a*, *b*, and *c*. The perimeter of the triangle is simply the sum of the three side lengths. The getArea method is a bit trickier. There is a useful geometric formula named Heron's formula which says that the area of a triangle with sides of lengths is related to a value *s* equal to half the triangle's perimeter:

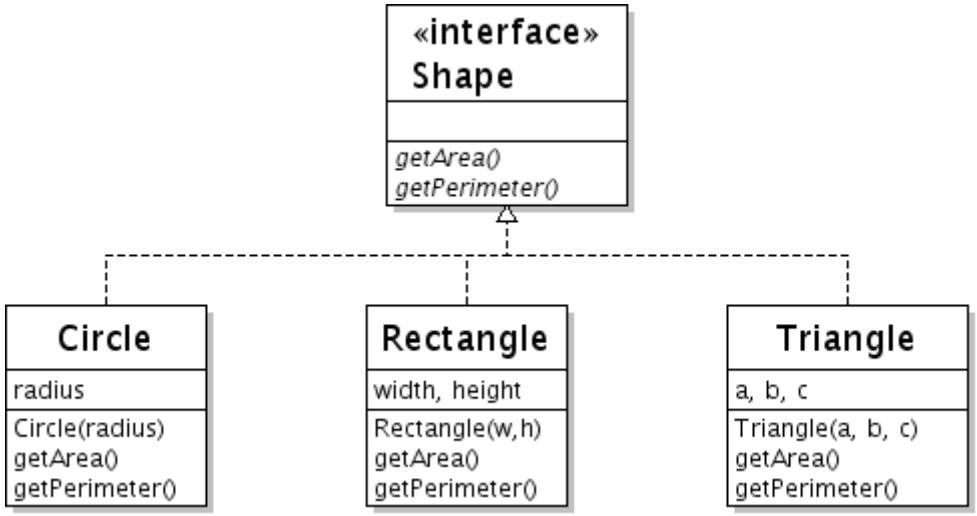
$$\text{area} = \sqrt{s\,(s-a)\,(s-b)\,(s-c)} \text{ where } s = \frac{a+b+c}{2}$$

Here is a complete version of the Triangle class:

```
// Represents triangle shapes.
public class Triangle implements Shape {
    private double a;
    private double b;
    private double c;

    // Constructs a new Triangle with sides of the given lengths.
    public Triangle(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // Returns the area of this triangle using Heron's formula.
    public double getArea() {
        double s = (a + b + c) / 2.0;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }

    // Returns the perimeter of this triangle.
    public double getPerimeter() {
        return a + b + c;
    }
}
```

Benefits of Interfaces

Classes that implement a common interface form a type hierarchy similar to those created by inheritance. The interface serves as a parent type for the classes that implement it. After our modifications, the type hierarchy looks like this:



The major benefit of interfaces is that we can use them to achieve polymorphism. We can now create an array of Shapes, pass a Shape as a parameter to a method, return a Shape from a method, and so on. The following program uses the shape classes in an example program similar to the polymorphism exercises in Section 9.3.

```
1  public class ShapesMain {
2      public static void main(String[] args) {
3          Shape[] shapes = new Shape[3];
4          shapes[0] = new Rectangle(18, 18);
5          shapes[1] = new Triangle(30, 30, 30);
6          shapes[2] = new Circle(12);
7
8          for (int i = 0; i < shapes.length; i++) {
9              System.out.println("area=" + shapes[i].getArea() +
10                                 ", perimeter=" + shapes[i].getPerimeter());
11         }
12     }
13 }
```

This program produces the following output:

```
area=324.0, perimeter=72.0
area=389.7114317029974, perimeter=90.0
area=452.3893421169302, perimeter=75.39822368615503
```

It may seem odd that we can have interface variables, arrays, and parameters when it isn't possible to construct an object of an interface type. It simply means that any object of a type that implements that interface may be used. In our case, any type that implements Shape (such as Circle, Rectangle, or Triangle) may be used.

Also recall that interfaces help us cope with the limitations of single inheritance. A class may only extend one superclass but may implement arbitrarily many interfaces. The following is the a general syntax for headers of classes that extend a superclass and implement one or more interfaces:

```
public class <name> extends <superclass name>
    implements <interface name>, <interface name>, ..., <interface name> {
    ...
}
```

There are many classes in the Java class libraries that both extend a superclass and implement one or more interfaces. For example, the PrintStream class (of which System.out is an instance) has the following header:

```
public class PrintStream extends FilterOutputStream
        implements Appendable, Closeable, Flushable
```

Interfaces in the Java Class Libraries

Interfaces are used in many other places in Java's class libraries. Here are just a few of Java's important interfaces:

- The `ActionListener` interface in the `java.awt` package is used to assign behavior to events when a user clicks on a button or other graphical control.
- The `Serializable` interface in the `java.io` package denotes classes whose objects are able to be saved to files and transferred over a network.
- The `Comparable` interface allows you to describe how to compare objects of your type to see which are less than, greater, or equal to each other. This can be used to search or sort a collection of your objects.
- The `Formattable` interface lets objects describe different ways they can be printed by the `System.out.printf` command.
- The `Runnable` interface is used for multi-threading, which allows a program to execute two pieces of code at the same time.
- Interfaces such as `List`, `Set`, `Map`, and `Iterator` in the `java.util` package describe data structures you can use to store collections of objects.

We will cover some of the preceding interfaces in later chapters.

9.7 Case Study: Designing a Hierarchy of Financial Classes

As you write larger and more complex programs, you will end up with more classes and more opportunities to use inheritance and interfaces. It is important to get practice devising sensible hierarchies of types, so that you will be able to solve large problems by breaking them down into good classes in the future.

When designing an object-oriented system, you should ask yourself the following questions:

- What classes of objects should I write?
- What behavior does the client wish each of these objects to have?
- What data do the objects need to store in order to implement this behavior?
- Are the classes related? If so, what is the nature of the relationships?

Having good answers to the above questions, along with a good knowledge of the necessary Java syntax, is a good start toward designing an object-oriented system. Such a process is called *object-oriented design*.

Object-Oriented Design (OOD)

Modeling a program or system as a collection of cooperating objects and individual objects, which are treated as instances of classes within class hierarchies.

Let's consider the problem of gathering information about a person's financial investments. We already have explored a Stock example in this chapter and the previous chapter. There are also stocks that pay dividends, which our DividendStock was written to handle. But Stocks are not the only type of financial asset that investors might have in their financial portfolios. Other investments might include mutual funds, real estate, or cash.

How should we design this system? What new types of objects should we write? Take a moment to consider it yourself. We'll discuss an example design next.

Class Design

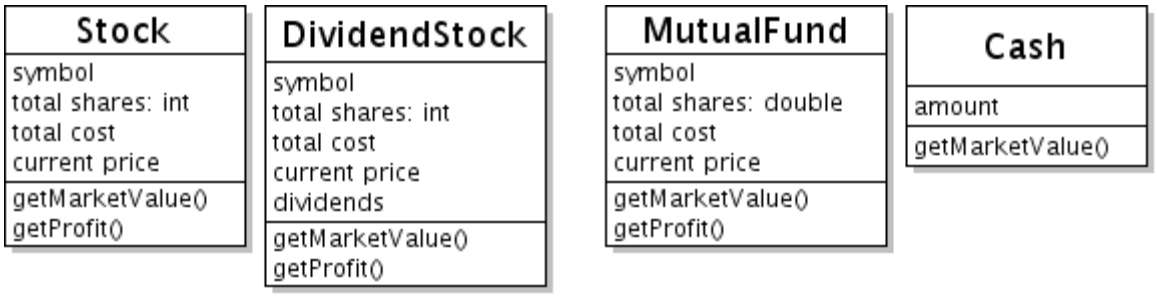
Each type of asset deserves its own class. We have the Stock class from the last chapter and its DividendStock subclass from earlier in this chapter. We can add classes like MutualFund and Cash. Each object of each of these types will represent a single investment of that type: for example, a MutualFund object will represent a purchase of a mutual fund, and a Cash object will represent a sum of money in the user's portfolio.

What data and behavior is necessary in each of these types of objects? Take a moment to consider it.

In terms of behavior, though each type of asset is unique, the types do have some common behavior. Each asset should be able to compute its current market value and profit, if any. These values are computed in different ways for different asset types. A stock's market value is the total number of shares purchased times the current price per share. Real estate's market value might depend on the current interest rate. Cash is always worth exactly its own amount.

In terms of data, we decided in the previous chapter that a Stock object should store the stock's symbol, the number of shares, the total cost paid for all shares, and the current price of the stock. Dividend stocks also need to store an amount of dividends paid. A MutualFund object stores the same, but mutual funds can hold partial shares. Cash only needs to store its amount.

We might update our diagram of types to reflect the preceding data and behavior:



Are the asset types related? It seems so. Perhaps we'd want to gather and store a person's portfolio of assets in an array. It would be convenient to be able to treat any asset the same way, insofar as they share similar functionality. For example, every asset has a market value, so it would be nice to compute the sum market value of all assets in an investor's portfolio array, without worrying about the exact type of each asset.

One hurdle is that different assets compute their market value in different ways. Stocks' values are based on the current share price, while the value of cash depends only on how much cash the person has. This implies that we should use an interface to represent the notion of an asset, and have every class previously named implement the asset interface. Our interface will demand that all assets have methods to get the market value and profit. The interface is a way of saying, "classes that want to consider themselves assets must have a getMarketValue and getProfit method." Our interface for financial assets would be saved into a file named Asset.java and might look like this:

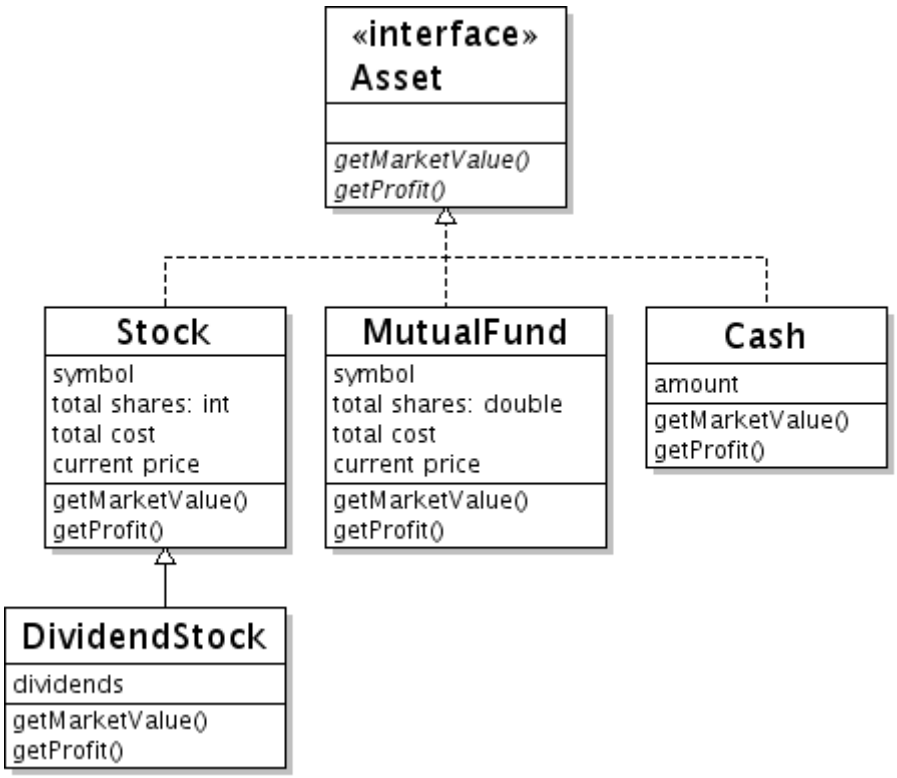
```
1  // This interface represents financial assets that investors hold.
2  public interface Asset {
3      // how much the asset is worth
4      public double getMarketValue();
5
6      // how much money has been made on this asset
7      public double getProfit();
8  }
```

We'll have our various classes certify that they are Assets by making them implement the Asset interface. For example, let's look at the Cash class. We didn't write a getProfit method in our previous diagram of Cash, because cash doesn't really change value and therefore doesn't have a profit. But we can just write a getProfit for Cash that returns 0.0 to indicate no profit.

```
1  // Represents an amount of money held by an investor.
2  public class Cash implements Asset {
3      private double amount;   // amount of money held
4
5      // Constructs a cash investment of the given amount.
6      public Cash(double amount) {
7          this.amount = amount;
8      }
9
10     // Returns this cash investment's market value, which is
11     // equal to the amount of cash.
12     public double getMarketValue() {
13         return amount;
14     }
15
16     // Since cash is a fixed asset, it never has a profit or loss.
17     public double getProfit() {
18         return 0.00;
19     }
20
21     // Sets the amount of cash invested to the given value.
22     public void setAmount(double amount) {
23         this.amount = amount;
24     }
25 }
```

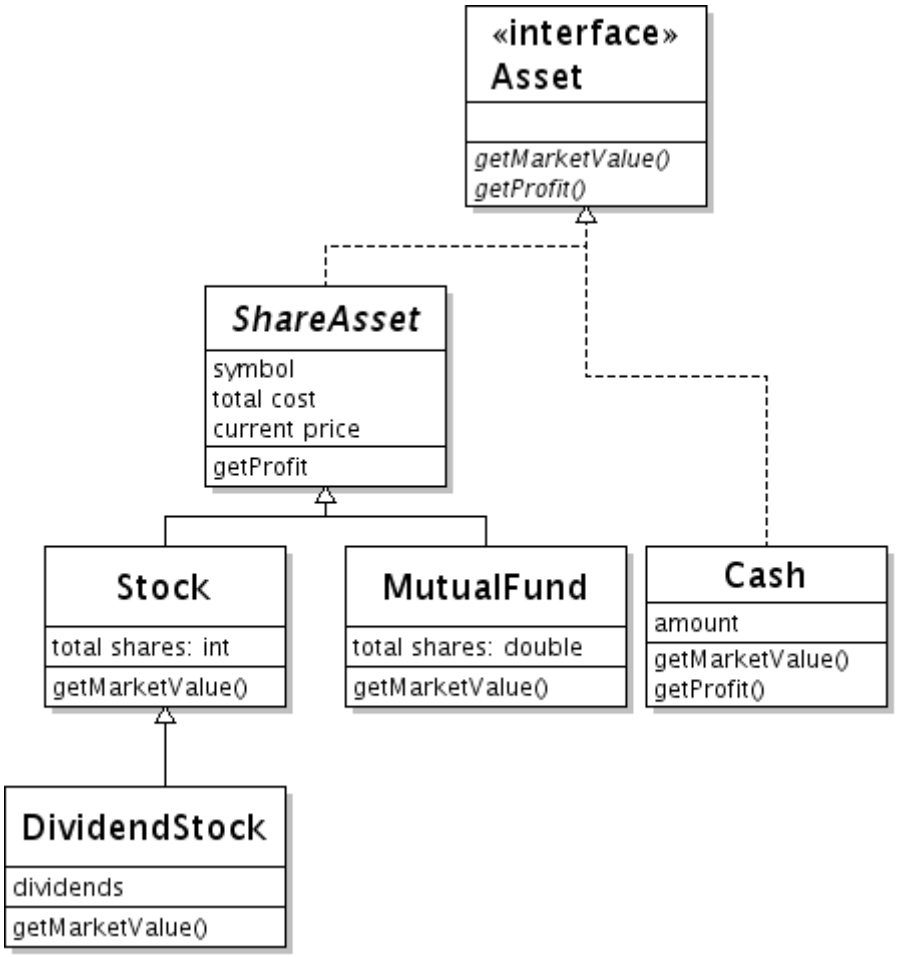
As discussed earlier in the chapter, a DividendStock is very similar to a normal Stock, only with a small amount of behavior added. Let's display DividendStock as a subclass of Stock through inheritance, so that code won't be redundant.

Our type hierarchy would now look like this:



What about the similarity between mutual funds and stocks? They both store assets based on shares, with a symbol, total cost, and current price. It wouldn't work very well to make one of them the subclass of the other, because the type of shares (integer or real number) isn't the same, and also because it's not a sensible 'is-a' relationship: stocks aren't really mutual funds and vice versa.

Let's try a different alternative by making a new common superclass named ShareAsset, representing any asset that has shares, that contains the common behavior of Stock and MutualFund. Then we can have both Stock and MutualFund extend ShareAsset, to reduce redundancy. In order to implement the parameterless getMarketValue method from the Asset interface, we'll add the current price as a data field, along with methods to get and set its value.



Initial Redundant Implementation

Here's some potential code for the ShareAsset class:

```
1  // Represents a general asset that has a symbol and holds shares.
2  public class ShareAsset {
3      private String symbol;
4      private double totalCost;
5      private double currentPrice;
6
7      // Constructs a new share asset with the given symbol and current price.
8      public ShareAsset(String symbol, double currentPrice) {
9          this.symbol = symbol;
10         this.currentPrice = currentPrice;
11         totalCost = 0.0;
12     }
13
14     // Adds a cost of the given amount to this asset.
15     public void addCost(double cost) {
16         totalCost += cost;
17     }
18
19     // Returns the given price per share of this asset.
20     public double getCurrentPrice() {
21         return currentPrice;
22     }
23
24     // Returns this asset's total cost.
25     public double getTotalCost() {
26         return totalCost;
27     }
28
29     // Sets the current share price of this asset to the given amount.
30     public void setCurrentPrice(double currentPrice) {
31         this.currentPrice = currentPrice;
32     }
33 }
```

We stole some code from last chapter's Stock class, but to fit this interface, the code underwent a few changes. Our old Stock code asked for the current share price as a parameter to its getProfit method. Since the getProfit method cannot accept any parameters if we wish to implement the interface, we'll instead store the current share price as a data field in the ShareAsset object. We supply a setCurrentPrice mutator method that can be called to set its proper value. We also give a constructor that can initialize with any number of shares and total cost.

One last modification we made to ShareAsset was to include an addCost method, which we'll use to add a given amount to the asset's total cost. We will need this because purchases on Stocks and MutualFunds need to update the totalCost data field, but they cannot do so directly since it is private.

The Stock class can now extend ShareAsset to implement its remaining functionality. Notice that we both extend ShareAsset and implement the Asset interface.

```
1  // A Stock object represents purchases of shares of a particular stock.
2  public class Stock extends ShareAsset implements Asset {
3      private int totalShares;
4
5      // Constructs a new Stock with the given symbol and current
6      // price per share.
7      public Stock(String symbol, double currentPrice) {
8          super(symbol, currentPrice);  // call Asset c'tor
9          totalShares = 0;
10     }
11
12     // Returns the market value of this stock, which is
13     // equal to the number of total shares times the share price.
14     public double getMarketValue() {
15         return totalShares * getCurrentPrice();
16     }
17
18     // Returns the total number of shares purchased.
19     public int getTotalShares() {
20         return totalShares;
21     }
22
23     // Returns the profit made on this stock.
24     public double getProfit() {
25         return getMarketValue() - getTotalCost();
26     }
27
28     // Records a purchase of the given number of shares of
29     // stock at the given price per share.
30     public void purchase(int shares, double pricePerShare) {
31         totalShares += shares;
32         addCost(shares * pricePerShare);
33     }
34 }
```

The MutualFund class receives a similar treatment but with a double for its total shares. (The two classes are awfully redundant; we'll improve them in the next section.)

```
1  // Represents a mutual fund asset.
2  public class MutualFund extends ShareAsset implements Asset {
3      private double totalShares;
4
5      // Constructs a new MutualFund investment with the given
6      // symbol and price per share.
7      public MutualFund(String symbol, double currentPrice) {
8          super(symbol, currentPrice);  // call Asset constructor
9          totalShares = 0.0;
10     }
11
12     // Returns the market value of this mutual fund, which
13     // is equal to the number of shares times the price per share.
14     public double getMarketValue() {
15         return totalShares * getCurrentPrice();
16     }
17
18     // Returns the number of shares of this mutual fund.
19     public double getTotalShares() {
20         return totalShares;
21     }
22
23     // Returns the profit made on this mutual fund.
24     public double getProfit() {
25         return getMarketValue() - getTotalCost();
26     }
27
28     // Records a purchase of the given number of shares for the
29     // given price per share.
30     public void purchase(double shares, double pricePerShare) {
31         totalShares += shares;
32         addCost(shares * pricePerShare);
33     }
34 }
```

The DividendStock simply adds an amount of dividend payments to a normal Stock, which affects its market value. The getProfit method doesn't need to be overridden in DividendStock, because DividendStock already inherits a getProfit method with the following body:

```
return getMarketValue() - getTotalCost();
```

Notice that getProfit's body calls getMarketValue. Since we overrode the getMarketValue method in DividendStock, a convenient side effect is that any other method that calls getMarketValue (such as getProfit) will also behave differently. This occurs because of polymorphism; since getMarketValue is overridden, getProfit calls the new version of the method. The profit will be correctly computed with dividends because these are added to the market value.

The following code implements the DividendStock class:

```
1  // A DividendStock object models shares of stocks with dividend payments.
2  public class DividendStock extends Stock {
3      private double dividends;   // amount of dividends paid
4
5      // Constructs a new DividendStock with the given symbol and price.
6      public DividendStock(String symbol, double currentPrice) {
7          super(symbol, currentPrice);  // call Stock constructor
8          dividends = 0.00;
9      }
10
11     // Returns this DividendStock's market value, which is
12     // a normal stock's market value plus any dividends.
13     public double getMarketValue() {
14         return super.getMarketValue() + dividends;
15     }
16
17     // Records a dividend payment of the given amount.
18     public void payDividend(double amountPerShare) {
19         dividends += amountPerShare * getTotalShares();
20     }
21 }
```

Abstract Classes

If inheritance lets us specify classes as parents and children, interfaces may be like distant cousins: related, but without many details in common. There are situations where we want more of a sibling relationship--classes with common method names, like we get with interfaces, but also with some code shared between them, like we get with inheritance.

One thing you may notice about the Stock and MutualFund code in the last section is that there is still a lot of redundancy. For example, the getMarketValue and getProfit methods, while they have identical code, can't be moved up into the ShareAsset superclass because they depend on the number of shares, which is different in each child class. We'd like to get rid of the redundancy somehow.

There is another problem. A ShareAsset isn't really a type of asset that a person can buy; it's just a concept that happens to be represented in our code. It would be undesirable if a person actually tried to construct a ShareAsset object--we only wrote that class to eliminate redundancy, not to be used directly.

If we want to create classes that shouldn't be used directly by outside code, but should only serve as a superclass for inheritance, we can designate the class as abstract. Writing `abstract` in the class's header will specify that a class cannot be used directly to construct objects.

```
public abstract class ShareAsset implements Asset {
    ...
}
```

An *abstract class* is one that captures common code to avoid redundancy, without introducing a new instantiable type into a system. An attempt to create a ShareAsset object will produce a compiler error such as the following, if ShareAsset was declared to be abstract:

```
C:\document\StockManager.java:55: ShareAsset is abstract; cannot be instantiated
      ShareAsset asset = new ShareAsset("MSFT", 20, 27.46);
      ^
1 error
```

Abstract Class

A Java class that serves only as a superclass for other classes but cannot actually be used to construct objects of its type.

A interesting thing about abstract classes is that they can claim to implement an interface without actually writing all of the methods in that interface. This 'passes the buck' to any subclass that extends the abstract class, requiring the subclass to finish writing the remaining methods.

A strange benefit of this is that the abstract class can actually call any of the interface's methods on itself, even if they don't exist in that file, because it can count on its subclasses to write the remaining methods. For example, if we make ShareAsset abstract and declare that it implements Asset, the header looks like the following:

```
public abstract class ShareAsset implements Asset {
```

The general syntax for declaring an abstract class is the following:

```
public abstract class <name> {
    ...
}
```

Now that ShareAsset is an Asset, we can move the common redundant getProfit code up to ShareAsset and out of Stock and MutualFund. ShareAsset objects can now call getMarketValue, even though that method isn't present in ShareAsset. The code compiles because the compiler knows that whatever class extends ShareAsset will have to implement getMarketValue.

```
// This method calls an abstract getMarketValue method
// (the subclass will provide its implementation)
public double getProfit() {
    return getMarketValue() - totalCost;
}
```

If necessary, abstract classes can also behave like interfaces and can demand that certain methods be implemented by all of their subclasses. Such required subclass behaviors are called abstract methods. If we hadn't had the Asset interface, but still wanted every ShareAsset subclass to have a getMarketValue method, we could have declared an abstract getMarketValue method:

```
public abstract double getMarketValue();
```

An abstract method is declared with a header similar to regular methods, but with the keyword `abstract` after the public modifier. The other difference about abstract methods is that they do not have method bodies--they only have a semicolon, which means that the behavior is specified in each subclass. An abstract class can contain both abstract and non-abstract methods. Technically,

every method in an interface is abstract as well. The methods in an interface can be declared with the `abstract` keyword if so desired, but the presence or absence of the abstract keyword has no effect.

The general syntax for abstract method declarations is the following:

```
public abstract <type> <name>(<type> <name>, ..., <type> <name>);
```

Because an abstract class can contain normal data fields and methods with implementation, and it can also contain abstract methods without implementation, abstract classes can be thought of as hybrids between classes and interfaces. One important difference between interfaces and abstract classes is that a class may choose to implement arbitrarily many interfaces, but it can only extend one abstract class.

We can modify the Stock and MutualFund classes to take advantage of ShareAsset and reduce the redundancy. Notice that we must implement getMarketValue, or else we receive an error. Here are the final versions of all three classes:

```
1  // Represents a general asset that has a symbol and holds shares.
2  public abstract class ShareAsset implements Asset {
3      private String symbol;
4      private double totalCost;
5      private double currentPrice;
6
7      // Constructs a new share asset with the given symbol and current price.
8      public ShareAsset(String symbol, double currentPrice) {
9          this.symbol = symbol;
10         this.currentPrice = currentPrice;
11         totalCost = 0.0;
12     }
13
14     // Adds a cost of the given amount to this asset.
15     public void addCost(double cost) {
16         totalCost += cost;
17     }
18
19     // Returns the given price per share of this asset.
20     public double getCurrentPrice() {
21         return currentPrice;
22     }
23
24     // This method calls an abstract getMarketValue method
25     // (the subclass will provide its implementation)
26     public double getProfit() {
27         return getMarketValue() - totalCost;
28     }
29
30     // Returns this asset's total cost.
31     public double getTotalCost() {
32         return totalCost;
33     }
34
```

```
    // Sets the current share price of this asset to the given amount.
    public void setCurrentPrice(double currentPrice) {
        this.currentPrice = currentPrice;
    }
}
```

```
// A Stock object represents purchases of shares of a particular stock.
public class Stock extends ShareAsset implements Asset {
    private int totalShares;

    // Constructs a new Stock with the given symbol and current
    // price per share.
    public Stock(String symbol, double currentPrice) {
        super(symbol, currentPrice);  // call Asset c'tor
        totalShares = 0;
    }

    // Returns the market value of this stock, which is
    // equal to the number of total shares times the share price.
    public double getMarketValue() {
        return totalShares * getCurrentPrice();
    }

    // Returns the total number of shares purchased.
    public int getTotalShares() {
        return totalShares;
    }

    // Records a purchase of the given number of shares of
    // stock at the given price per share.
    public void purchase(int shares, double pricePerShare) {
        totalShares += shares;
        addCost(shares * pricePerShare);
    }
}
```

```
1  // Represents a mutual fund asset.
2  public class MutualFund extends ShareAsset {
3      private double totalShares;
4
5      // Constructs a new MutualFund investment with the given
6      // symbol and price per share.
7      public MutualFund(String symbol, double currentPrice) {
8          super(symbol, currentPrice);  // call Asset constructor
9          totalShares = 0.0;
10     }
11
12     // Returns the market value of this mutual fund, which
13     // is equal to the number of shares times the price per share.
14     public double getMarketValue() {
15         return totalShares * getCurrentPrice();
16     }
17
18     // Returns the number of shares of this mutual fund.
19     public double getTotalShares() {
20         return totalShares;
21     }
22
23     // Records a purchase of the given number of shares for the
24     // given price per share.
25     public void purchase(double shares, double pricePerShare) {
26         totalShares += shares;
27         addCost(shares * pricePerShare);
28     }
29 }
```

We have achieved a solid object-oriented design for these tricky investment types, which can now be used by other classes to manage the user's finances.

Chapter Summary

- Inheritance allows the creation of a parent-child relationship between two types.
- The child class of an inheritance relationship (commonly called a subclass) will receive a copy of ("inherit") every field and method from the parent class (superclass). The subclass "extends" the superclass, because it can add new fields and methods to the ones it inherits from the superclass.
- A subclass can override methods from the superclass by writing its own version of them, which will replace the one that was inherited.
- Treating objects of different types interchangeably is called polymorphism.
- Subclasses can refer to the superclass's constructors or methods using the `super` keyword.
- An interface is a list of method declarations. An interface specifies method names, parameters, and return types, but does not include the bodies of the methods.
- Classes can promise to implement all of the methods in an interface. Implementing an interface is done with the `implements` keyword. The advantage of interfaces is that we can use them to achieve polymorphism, so that we can treat several different classes the same way.

- If two or more classes both implement the same interface, we can use either of them interchangeably and call any of the interface's methods on them.
- An abstract class is a class that can not be instantiated. No objects of the abstract type can be constructed. An abstract class is useful because it can be used as a superclass and extended.
- An abstract class can contain abstract methods, which are declared but do not have a body. All subclasses of an abstract class must implement the abstract superclass's abstract methods.

Self-Check Problems

Section 9.1: Inheritance Concepts

1. What is code reuse? How does inheritance help achieve code reuse?

Section 9.2: Programming with Inheritance

2. What is the difference between overloading and overriding a method?
3. Consider the following classes:

```
public class Vehicle { ... }

public class Car extends Vehicle { ... }

public class SUV extends Car { ... }
```

Which of the following are legal statements?

- `Vehicle v = new Car();`
- `Vehicle v = new SUV();`
- `Car c = new SUV();`
- `SUV s = nev SUV();`
- `SUV s = nev Car();`
- `Car c = new Vehicle();`

Section 9.3: The Mechanics of Polymorphism

4. Using the A, B, C, and D classes from this section, what is the output of the following code fragment?

```
public static void main(String[] args) {
    A[] elements = {new B(), new D(), new A(), new C()};
    for (int i = 0; i < elements.length; i++) {
        elements[i].method2();
        System.out.println(elements[i]);
        elements[i].method1();
        System.out.println();
    }
}
```

5. Assume the following classes have been defined:

```
1  public class Flute extends Blue {
2      public void method2() {
3          System.out.println("flute 2");
4      }
5
6      public String toString() {
7          return "flute";
8      }
9  }
```

```
1  public class Blue extends Moo {
2      public void method1() {
3          System.out.println("blue 1");
4      }
5  }
```

```
1  public class Shoe extends Flute {
2      public void method1() {
3          System.out.println("shoe 1");
4      }
5  }
```

```
1  public class Moo {
2      public void method1() {
3          System.out.println("moo 1");
4      }
5
6      public void method2() {
7          System.out.println("moo 2");
8      }
9
10     public String toString() {
11         return "moo";
12     }
13 }
```

What is the output produced by the following code fragment?

```
public static void main(String[] args) {
    Moo[] elements = {new Shoe(), new Flute(), new Moo(), new Blue()};
    for (int i = 0; i < elements.length; i++) {
        System.out.println(elements[i]);
        elements[i].method1();
        elements[i].method2();
        System.out.println();
    }
}
```

6. Using the same classes from the previous problem, what is the output produced by the following code fragment?

```
public static void main(String[] args) {
    Moo[] elements = {new Blue(), new Moo(), new Shoe(), new Flute()};
    for (int i = 0; i < elements.length; i++) {
        elements[i].method2();
        elements[i].method1();
        System.out.println(elements[i]);
        System.out.println();
    }
}
```

7. Assume the following classes have been defined:

```
1  public class Mammal extends SeaCreature {
2      public void method1() {
3          System.out.println("warm-blooded");
4      }
5  }
```

```
1  public class SeaCreature {
2      public void method1() {
3          System.out.println("creature 1");
4      }
5
6      public void method2() {
7          System.out.println("creature 2");
8      }
9
10     public String toString() {
11         return "ocean-dwelling";
12     }
13 }
```

```
1  public class Whale extends Mammal {
2      public void method1() {
3          System.out.println("spout");
4      }
5
6      public String toString() {
7          return "BIG!";
8      }
9  }
```

```
1  public class Squid extends SeaCreature {
2      public void method2() {
3          System.out.println("tentacles");
4      }
5
6      public String toString() {
7          return "squid";
8      }
9  }
```

What is the output produced by the following code fragment?

```
public static void main(String[] args) {
    SeaCreature[] elements = {new Squid(), new Whale(), new SeaCreature(),
                              new Mammal()};
    for (int i = 0; i < elements.length; i++) {
        System.out.println(elements[i]);
        elements[i].method1();
        elements[i].method2();
        System.out.println();
    }
}
```

8. Using the same classes from the previous problem, what is the output produced by the following code fragment?

```
public static void main(String[] args) {
    SeaCreature[] elements = {new SeaCreature(), new Squid(),
                              new Mammal(), new Whale()};
    for (int i = 0; i < elements.length; i++) {
        elements[i].method2();
        System.out.println(elements[i]);
        elements[i].method1();
        System.out.println();
    }
}
```

Section 9.4: Interacting with the Superclass

9. Explain the difference between the `this` keyword and the `super` keyword. When should each be used?

10. Consider the following two classes:

```
public class Student {
    private String name;
    private int age;

    ...

    public int setAge(int age) {
        this.age = age;
    }
}

public class Freshman extends Student {
    ...
}
```

Can the code in the Freshman class access the name and age data fields it inherits from Student? Can it call the setAge method?

11. Add the following constructor to the Point3D class shown in this section:

    `public Point3D()`
    Constructs a new 3D point at the origin of (0, 0, 0). Use the `super` keyword or the `this` keyword as part of your solution.

Section 9.6: Interfaces

12. What is the difference between implementing an interface and extending a class?

13. Consider the following interface and class:

```
public interface I {
    public void m1();
    public void m2();
}

public class C implements I {
    // code for class C
}
```

What must be true about the code for class C, in order for it to compile successfully?

14. What's wrong with the code for the following interface? What should be changed to make a valid interface for objects that have a color?

```
public interface Colored {
    private Color color;

    public Color getColor() {
        return this.color;
    }
}
```

15. Modify the Point class from Chapter 8 so that it has a color and implements the Colored interface. (You may wish to create a ColoredPoint class that extends Point.)

Section 9.7: Case Study: Designing a Hierarchy of Financial Classes

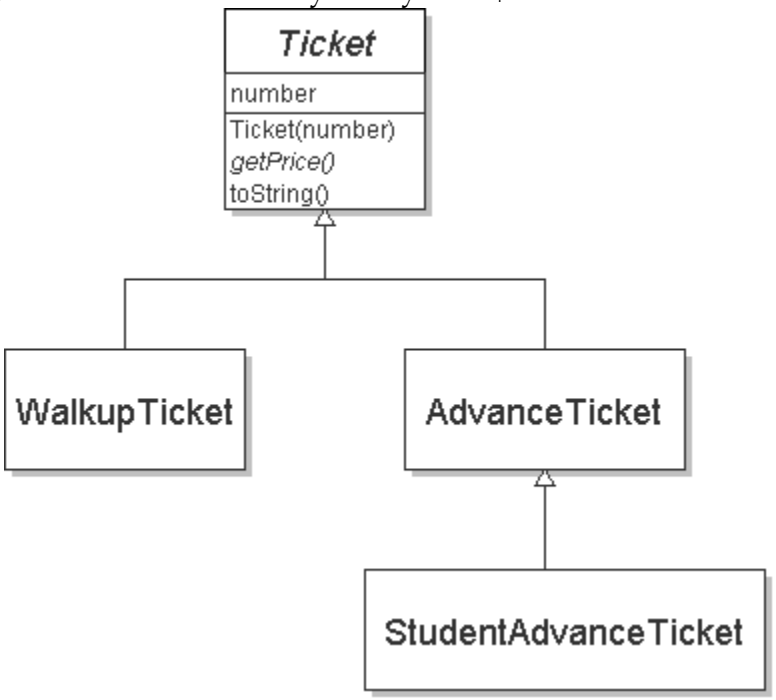
16. What is an abstract class? How is an abstract class like a normal class, and how does it differ? How is it like an interface?

17. Consider writing a program to be used to manage a collection of movies. There are three kinds of movies in the collection: dramas, comedies, and documentaries. The collector would like to keep track of each movie's name, the name of the director, and the date when it was made. Some operations are to be implemented for all movies, and there will also be special operations for each of the three different kinds of movies. How would you design the class(es) to represent this system of movies?

Exercises

For the next four problems, consider the task of representing types of tickets to school campus events. Each ticket has a unique number and a price. There are three types of tickets: walk-up tickets, advance tickets, and student advance tickets.

- Walk-up tickets are purchased the day of the event and cost $50.
- Advance tickets purchased 10 or more days before the event cost $30, and advance tickets purchased fewer than 10 days before the event cost $40.
- Student advance tickets are sold at half the price of normal advance tickets: ones 10 days early cost $15, and fewer than 10 days early cost $20.



1. Implement a class named `Ticket` that will serve as the superclass for all three types of tickets. Define all common operations in this class, and specify all differing operations in such a way that every subclass must implement them. No actual objects of type `Ticket` will be created: each actual ticket will be an object of a subclass type. Define the following operations:
   - The ability to construct a ticket by number.

- The ability to ask for a ticket's price.
- The ability to `println` a ticket object as a `String`. An example string would be `"Number: 17, Price: 50.0"`.

2. Implement a class named `WalkupTicket` to represent a walk-up event ticket. Walkup tickets are also constructed by number, and they have a price of $50.00.

3. Implement a class named `AdvanceTicket` to represent tickets purchased in advance. An advance ticket is constructed with a ticket number and also how many days in advance the ticket was purchased. Advance tickets purchased 10 or more days before the event cost $30, and advance tickets purchased fewer than 10 days before the event cost $40.

4. Implement a class named `StudentAdvanceTicket` to represent tickets purchased in advance by students. A student advance ticket is constructed with a ticket number and also how many days in advance the ticket was purchased. Student advance tickets purchased 10 or more days before the event cost $15, and advance tickets purchased fewer than 10 days before the event cost $20 (half of a normal advance ticket). When a student advance ticket is printed, the string should mention that the student must show their student ID. For example, `"Number: 17, Price: 15.0 (ID required)"`.

For the next six problems, consider the task of representing types of birds in an aviary simulation. Each bird has an (x, y) position and a color, and each bird can fly. Different types of birds have different flight behavior.

5. Write an interface named `Bird` to represent different types of birds. The interface should have methods to do the following:

   - Get the bird's color.
   - Get the bird's (x, y) position as a Point.
   - Tell a bird to fly. (Each time a bird is told to fly, it will move its position once.)

   Your `Bird` interface should work with the following client program:

```
1  import java.awt.*;
2
3  public class Aviary {
4      public static final int SIZE = 20;
5      public static final int PIXELS = 10;
6
7      public static void main(String[] args) {
8          // create a drawing panel
9          DrawingPanel panel = new DrawingPanel(SIZE * PIXELS, SIZE * PIXELS);
10         Graphics g = panel.getGraphics();
11
12         // create several birds
13         Bird[] birds = {
14                 new Cardinal(7, 4),
15             new Cardinal(3, 8),
16             new Hummingbird(2, 9),
17             new Hummingbird(16, 11),
18             new Bluebird(4, 15),
19             new Bluebird(8, 1),
20             new Vulture(3, 2),
21             new Vulture(18, 14),
22         };
23
24         while (true) {
25             // clear screen
26             g.setColor(Color.WHITE);
27             g.fillRect(0, 0, SIZE * PIXELS, SIZE * PIXELS);
28
29             // tell each bird to fly, and draw the bird in its new position
30             for (Bird bird : birds) {
31                 bird.fly();
32                 g.setColor(bird.getColor());
33                 Point pos = bird.getPosition();
34                 g.fillOval(pos.getX() * PIXELS, pos.getY() * PIXELS, PIXELS, PIXE
35             }
36
37             panel.sleep(500);
38         }
39     }
40 }
```

6. Write a class named `Cardinal` that represents cardinal birds. A cardinal is red in color. The cardinal's movement is vertical. Initially a cardinal is moving up. Each time the cardinal is told to fly, it will move its position one unit upward on the y-axis (remember that upward is negative). If the cardinal hits the edge of the aviary (a y-coordinate of 0 or 19), it turns around and flies in the opposite direction.

   Assume that the aviary's size of 20 is stored in a static constant named `Aviary.SIZE`. You may wish to introduce an abstract class to hold behavior that will be common to all bird classes.

7. Write a class named `Hummingbird` that represents hummingbirds. A hummingbird is magenta in color. The hummingbird's movement is random; each time the hummingbird is told to fly, it will pick a new random (x, y) position in the range of (0, 0) to (19, 19).

8. Write a class named `Bluebird` that represents bluebirds. A bluebird is blue in color. The bluebird's movement is in a zig-zag pattern. Initially the bluebird faces right. The bluebird moves in an alternating pattern of up-right, down-right, up-right, down-right, and so on until it hits the right edge of the aviary (x-coordinate of 19), at which point it turns around. Subsequent calls to `fly` will cause the bird to move up-left, down-left, up-left, down-left, and so on until it hits the left edge of the aviary.

9. Write a class named `Vulture` that represents vultures. A vulture is black in color. The vulture's movement is in a counter-clockwise circle pattern. Initially the vulture faces up. The first time the vulture flies, it moves up by one, then it turns to face left. Its second move, it moves left by one and turns to face down. Its third move, it moves down and turns to face right. Its fourth move, it moves right by one and turns to face up. The pattern repeats in this fashion.

10. What changes should be made if we want to count the number of times each bird has flown?

Programming Projects

1. Write an inheritance hierarchy of 3-dimensional shapes. Make a top-level shape interface that has methods for getting information such as the volume and surface area of a 3D shape. Then make classes and subclasses that implement various shapes such as cubes, rectangular prisms, spheres, triangular prisms, and cylinders. Place common behavior in superclasses whenever possible, and use abstract classes as appropriate. Add methods to the subclasses to represent the unique behavior of each 3D shape, such as a method to get a sphere's radius.

2. Write a set of classes that define the behavior of certain animals. They can be used in a simulation of a world with many animals moving around in it. Different kinds of animals will move in different ways (you are defining those differences). As the simulation runs, animals can "die" by ending up on the same location, in which case the simulator randomly selects one animal to survive the collision.

   The behavior of each animal class is the following:

Class	getChar	getMove
Bird	B	Randomly selects one of the four directions each time
Frog	F	Picks a random direction, moves 3 in that direction, repeat (same as bird, but staying in a single direction longer)
Mouse	M	West 1, north 1, repeat (zig zag to the NW)
Turtle	T	South 5, west 5, north 5, east 5, repeat (clockwise box)
Wolf	W	You define this to have any custom behavior you want

   Your classes should be stored in files called Bird.java, Frog.java, Mouse.java, Turtle.java and Wolf.java.

3. Write an inheritance hierarchy that models sports players. Create a common superclass and/or interface to store information common to any player regardless of sport, such as name, number, and salary. Then create subclasses for players of your favorite sports such as basketball, soccer, or tennis. Place sport-specific information and behavior (such as kicking or vertical jump height) into subclasses whenever possible.

4. Write an inheritance hierarchy to model items at a library. Include books, magazines, journal articles, videos, and electronic media such as CDs. Include common information in a superclass and/or interface that the library must have for every item, such as a unique identification number and title. Place item-type-specific behavior and information, such as a video's runtime length or a CD's musical genre, into the subclasses.

Stuart Reges
Marty Stepp

Chapter 10
ArrayLists

Copyright © 2006 by Stuart Reges and Marty Stepp

- 10.1 ArrayLists
  - Basic ArrayList Operations
  - ArrayList Searching Methods
  - Sample ArrayList Problems
  - The for-each Loop
  - Wrapper Classes
- 10.2 The Comparable Interface
  - Natural Ordering and compareTo
  - Implementing Comparable
- 10.3 Case Study: Vocabulary Comparison
  - Version 1: Compute Vocabulary
  - Version 2: Compute Overlap
  - Version 3: Complete Program

Introduction

In Chapter 7 we saw how to define arrays that can be used to store a sequence of values all of the same type. In this chapter we explore a new structure known as an ArrayList that provides even more functionality than an array. Remember that arrays are fixed-size structures. ArrayLists have a variable length, which allows them to grow and shrink as the program executes.

The ArrayList structure is the first example of a generic structure, so we will have to discuss how generics work in Java. We will also see how to use primitive data with such a structure using what are known as the wrapper classes. Finally, we will see how the Comparable interface is used to describe how to put values of a particular type into sorted order and we will see how to write classes that implement the Comparable interface.

10.1 ArrayLists

Arrays are a powerful and important part of the Java language, but they aren't always easy to manipulate. For example, what if we want to store variable length data with a list of values that grows and shrinks as the program executes? This behavior isn't easy to implement with an array because arrays have a fixed size.

Java offers an alternative. The Java class libraries include a structure known as an ArrayList. As its name suggests, it is based on array technology. Each ArrayList uses an array to store its values. As a result, the ArrayList has the same fast random access that arrays provide. In addition, the ArrayList provides many methods that make it more convenient than an array for certain programming tasks.

If you read the API documentation for ArrayList, you'll see that it is actually listed as ArrayList<E>. This is an example of a *generic class* in Java.

Generic Class (Generic)

A class such as ArrayList<E> that takes a type parameter to indicate what kind of values will be used.

The "E" in ArrayList<E> is short for "Element" and it indicates the type of elements we will be including in the ArrayList. For example, we might construct an ArrayList<String> if we want to make a list of String objects or we can make an ArrayList<Point> if we want to make a list of Point objects or an ArrayList<Color> if we want a list of Color objects.

Basic ArrayList Operations

The syntax for constructing an ArrayList is more complicated than we have seen with other types because of the type parameter. For example, we would construct an ArrayList<String> as follows:

```
ArrayList<String> list = new ArrayList<String>();
```

This code constructs an empty ArrayList<String>. This syntax is complicated but it will be easier to remember if you keep in mind that the "<String>" notation is actually part of the type. This isn't simply an ArrayList, it is an ArrayList<String> (often read as "an ArrayList of String"). Notice how the type appears in declaring the variable and in calling the constructor:

```
ArrayList<String> list = new ArrayList<String>();
~~~~~~~~~~~~~~~~~            ~~~~~~~~~~~~~~~~~
      type                         type
```

If you think in terms of the type being ArrayList<String>, then really this line of code isn't all that different from the code we have been writing for an object like a Point:

```
Point p = new Point();
~~~~~         ~~~~~
type          type
```

Once an ArrayList has been constructed, we can add values to it by calling the add method:

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello");
list.add("world");
list.add("this is fun");
```

When you ask an ArrayList to add a new value to the list, it adds the new value to the end of the list. In other words, the simple version of add appends the value to the end of the list.

Unlike simple arrays, the ArrayList class overrides Java's toString method with something useful. The ArrayList version of toString constructs a String that includes the contents of the list inside of square brackets and separated by commas. Remember that the toString method is called when we print an object or concatenate the object to a String. As a result, ArrayLists can be printed with a simple println, as in:

```
System.out.println("list = " + list);
```

For example, we can add println statements as we add values to the list:

```
ArrayList<String> list = new ArrayList<String>();
System.out.println("list = " + list);
list.add("hello");
System.out.println("list = " + list);
list.add("world");
System.out.println("list = " + list);
list.add("this is fun");
System.out.println("list = " + list);
```

If we execute this code, we get the following output:

```
list = []
list = [hello]
list = [hello, world]
list = [hello, world, this is fun]
```

Notice that can print the ArrayList even when it is empty and that new values are added at the end. The ArrayList class also provides a method for adding values in the middle of the list. It preserves the order of other list elements, shifting values right to make room for the new value. To indicate positions within the list, we use the Java convention of indexing starting at 0 for the first value, 1 for the second value and so on. For example, given the list above, consider the effect of inserting a value at index 1:

```
list.add(1, "middle value");
System.out.println("now list = " + list);
```

The call on add instructs the computer to insert the new String at index 1. That means that the old value at index 1 and everything that comes after it gets shifted to the right. So the following output is produced:

```
now list = [hello, middle value, world, this is fun]
```

So now the list has a total of four values. Notice that this new method is also called add but it takes two parameters (an index and a value to insert). This is another example of overloading, which was discussed in Chapter 3.

The ArrayList also has a method for removing a value at a particular index. The remove method also preserves the order of the list by shifting values left to fill in any gap. For example, consider what happens given the list above if we remove the value at position 0 and then remove the value at position 1:

```
System.out.println("before remove list = " + list);
list.remove(0);
list.remove(1);
System.out.println("after remove list = " + list);
```

This code produces the following output:

```
before remove list = [hello, middle value, world, this is fun]
after remove list = [middle value, this is fun]
```

This result is a little surprising. We asked the list to remove the value at position 0 and then to remove the value at position 1. You might imagine that this would get rid of the Strings "hello" and "middle value" since they were at positions 0 and 1, respectively, before this code was executed. But you have to remember that an ArrayList is a dynamic structure where values are moving around and shifting into new positions. We can see this more clearly if we include a second println statement:

```
System.out.println("before remove list = " + list);
list.remove(0);
System.out.println("after 1st remove list = " + list);
list.remove(1);
System.out.println("after 2nd remove list = " + list);
```

This code produces the following output:

```
before remove list = [hello, middle value, world, this is fun]
after 1st remove list = [middle value, world, this is fun]
after 2nd remove list = [middle value, this is fun]
```

The first call on remove removes the String "hello" because it's the value currently in position 0. But once that value has been removed, everything else shifts over. So the String "middle value" moves to the front (to position 0) and the String "world" shifts into position 1. So when the second call on remove is performed, Java removes "world" from the list because it is the value that is in position 1 at that point in time.

If you want to find out how many elements are in an ArrayList, you can call its size method and if you want to obtain an individual item from the list, you can call its get method passing it a specific index. For example, the loop below would add up the lengths of the Strings in an ArrayList<String>.

```
int sum = 0;
for (int i = 0; i < list.size(); i++) {
    String s = list.get(i);
    sum += s.length();
}
System.out.println("Total of lengths = " + sum);
```

This loop looks similar to the kind of loop we would use to access the various elements of an array. But instead of asking for list.length as we would for an array, we ask for list.size() and instead of asking for list[i] as we would with an array, we ask for list.get(i).

Calling add and remove can be expensive because of the shifting of values. As a result, there is a method called set that takes an index and a value and that replaces the value at the given index with the given value without doing any shifting. For example, we can replace the value at the front of the list by saying:

```
list.set(0, "new front");
```

As noted above, when you construct an ArrayList it will initially be empty. After you have added values to a list, you can remove them one at a time. But sometimes you want to remove all of them. In that case, you can call the clear method of the ArrayList to have it remove all of the items from the list.

The ArrayList class is part of the java.util package, so to include it in a program, you would have to include an import declaration.

Below is a table summarizing the ArrayList operations we have seen in this section. They are defined in terms of the generic type "E". A more complete list can be found in the online Java documentation.

Basic ArrayList Methods

Method	Description	ArrayList<String> Example
`add(E value)`	adds value at the end of the list	`list.add("end");`
`add(int index, E value)`	adds value at the given index, shifting subsequent values right	`list.add(1, "middle");`
`clear()`	makes the list empty	`list.clear();`
`get(int index)`	gets the value at the given index (return type is E)	`list.get(1)`
`remove(int index)`	removes the value at the given index, shifting subsequent values left	`list.remove(1);`
`set(int index, E value)`	replaces the value at given index with given value	`list.set(2, "hello");`
`size()`	returns the current number of elements in the list	`list.size()`

ArrayList Searching Methods

Once you have built up an ArrayList, you might be interested in searching for a specific value in the list. The ArrayList class provides several mechanisms for doing so. If you just want to know whether or not something is in the list, you can call the method called contains that returns a boolean value. For example, suppose that you have an input file of names that has some duplicates and you want to get rid of the duplicates. For example, the file might store:

```
Maria Derek Erica
Livia Jack Anita
Kendall Maria Livia Derek
Jamie Jack
Erica
```

You can construct an ArrayList<String> to solve this problem by using the contains method to avoid adding any duplicates:

```
Scanner input = new Scanner(new File("names.txt"));
ArrayList<String> list = new ArrayList<String>();
while (input.hasNext()) {
    String name = input.hasNext();
    if (!list.contains(name)) {
        list.add(name);
    }
}
System.out.println("list = " + list);
```

Given the input file above, this code produces the following output:

```
list = [Maria, Derek, Erica, Livia, Jack, Anita, Kendall, Jamie]
```

Notice that only eight of the original thirteen names appear in this list because the various duplicates have been eliminated.

Sometimes it is not enough to know that a value appears in the list. Sometimes you want to know exactly where it occurs. For example, suppose you want to write a method to replace all occurrences of one word with another word in an ArrayList<String>. You can call the set method to replace each individual occurrence, but you have to know where they are in the list. You can find out the location of a value in the list by calling the indexOf method.

The indexOf method takes a value to search for. It returns the index of the first occurrence of the value in the list. If it doesn't find the value, it returns -1. So we could write the replace method as follows:

```
public static void replace(String target, String replacement,
                           ArrayList<String> list) {
    int index = list.indexOf(target);
    while (index >= 0) {
        list.set(index, replacement);
        index = list.indexOf(target);
    }
}
```

Notice that the return type of this method is void even though it changes the contents of an ArrayList object. Some novices think that you have to return the changed ArrayList, but the method doesn't create a new ArrayList, it merely changes the contents of the list. As we have seen with arrays and other objects, a parameter is all we need to be able to change the current state of an object.

We can test the method with the following code:

```
ArrayList<String> list = new ArrayList<String>();
list.add("to");
list.add("be");
list.add("or");
list.add("not");
list.add("to");
list.add("be");
System.out.println("initial list = " + list);
replace("be", "beep", list);
System.out.println("final list = " + list);
```

This code produces the following output:

```
initial list = [to, be, or, not, to, be]
final list = [to, beep, or, not, to, beep]
```

There is also a variation of indexOf known as lastIndexOf. As its name implies, it returns the index of the last occurrence of the value. This might be useful for algorithms that you want to run backwards.

The following table summarizes the ArrayList searching methods.

ArrayList Searching Methods

Method	Description	ArrayList<String> Example
`contains(Object value)`	returns true if the given value appears in the list, returns false otherwise	`list.contains("hello")`
`indexOf(E value)`	returns the index of the first occurrence of the given value in the list (-1 if not found)	`list.indexOf("world")`
`lastIndexOf(E value)`	returns the index of the last occurrence of the given value in the list (-1 if not found)	`list.lastIndexOf("hello")`

Sample ArrayList Problems

Let's solve two ArrayList problems to explore some of the issues that come up when using ArrayList objects. Consider the following code that creates an ArrayList and stores several words in it:

```
ArrayList<String> words = new ArrayList<String>();
words.add("four");
words.add("score");
words.add("and");
words.add("seven");
words.add("years");
words.add("ago");
System.out.println("words = " + words);
```

This code produces the following output:

```
words = [four, score, and, seven, years, ago]
```

Suppose that we want to insert a String containing a single asterisk in front of each word ("*"). We expect to double the length of the ArrayList because each of these words will have an asterisk in front of it if our code works properly. Here is a first attempt that makes sense intuitively:

```
for (int i = 0; i < words.size(); i++) {
    words.add(i, "*");
}
System.out.println("after loop words = " + words);
```

So let's get back to the for loop. It has the usual array-style for loop with an index variable i that starts at 0 and goes up by one each time. In this case it is inserting an asterisk at position i each time through the loop. The problem is that the loop never terminates. If you're patient enough you will find that the program does eventually terminate with an "out of memory" error.

The problem is the fact that the ArrayList is a dynamic structure where things move around to new positions. Let's think about this carefully to see what is going on. Initially we have this list with the String "four" in position 0:

```
[four, score, and, seven, years, ago]
```

The first time through the loop, we insert an asterisk at position 0. This shifts the String "four" one to the right, so that it is now in position 1.

```
[*, four, score, and, seven, years, ago]
```

Then we come around the for loop and increment i to be 1. And we insert an asterisk at position 1. But notice that the word "four" is currently at position 1, which means that this second asterisk also goes in front of the word "four", shifting it into position 2:

```
[*, *, four, score, and, seven, years, ago]
```

We go around the loop again, incrementing i to be 2 and once again inserting an asterisk at that position, which is once again in front of the word "four":

```
[*, *, *, four, score, and, seven, years, ago]
```

This continues indefinitely because we keep inserting asterisks in front of the first word in the list. The for loop test compares i to the size of the list, but because the list is growing, the size keeps going up. So this process continues until we exhaust all available memory.

To fix this loop, we have to realize that inserting an asterisk at position i is going to shift everything one to the right. So on the next iteration of the loop, we will want to deal with the position two to the right, not the position one to the right. So we can fix the loop simply by changing the update part of the for loop to add 2 to i instead of adding 1 to i:

```
for (int i = 0; i < words.size(); i += 2) {
    words.add(i, "*");
}
System.out.println("after loop words = " + words);
```

When we execute this version of the code, we get the following output:

```
after loop words = [*, four, *, score, *, and, *, seven, *, years, *, ago]
```

As another example, let's consider what code we would need to write to undo this operation. In particular, we want to write code that will remove every other value from the list starting with the first value. So we want to remove the values that are currently at indexes 0, 2, 4, 6, 8 and 10. That might lead us to write code like the following:

```
for (int i = 0; i < words.size(); i += 2) {
    words.remove(i);
}
System.out.println("after second loop words = " + words);
```

Looking at the loop you can see that i starts at 0 and goes up by 2 each time, which means it produces a sequence of even values (0, 2, 4 and so on). That would seem to be right given that the values to be removed are at those indexes. But this code doesn't work. It produces the following output:

```
after second loop words = [four, *, *, and, seven, *, *, ago]
```

Again, our problems comes from the fact that the ArrayList is a dynamic structure where values are shifted from one location to another. The first asterisk we want to remove is at index 0:

```
[*, four, *, score, *, and, *, seven, *, years, *, ago]
```

But once we remove the asterisk at position 0, the second asterisk shifts into index 1:

```
[four, *, score, *, and, *, seven, *, years, *, ago]
```

So the second remove should be at index 1, not index 2. And once we perform that second remove, the third asterisk will be in index 2:

```
[four, score, *, and, *, seven, *, years, *, ago]
```

So in this case, we don't want to increment i by 2 each time through the loop. Here the simple loop that increments by 1 is the right choice:

```
for (int i = 0; i < words.size(); i++) {
    words.remove(i);
}
System.out.println("after second loop words = " + words);
```

After executing this code, we obtain the following output:

```
after second loop words = [four, score, and, seven, years, ago]
```

Putting all of these pieces together, we obtain the following complete program:

```
1  // Builds up a list of words, adds stars to it and removes them.
2
3  import java.util.*;
4
5  public class StarFun {
6      public static void main(String[] args) {
7      // construct and fill up ArrayList
8      ArrayList<String> words = new ArrayList<String>();
9      words.add("four");
10     words.add("score");
11     words.add("and");
12     words.add("seven");
13     words.add("years");
14     words.add("ago");
15     System.out.println("words = " + words);
16
17     // insert one star in front of each word
18     for (int i = 0; i < words.size(); i += 2) {
19         words.add(i, "*");
20     }
21     System.out.println("after loop words = " + words);
22
23     // remove stars
24     for (int i = 0; i < words.size(); i++) {
25         words.remove(i);
26     }
27     System.out.println("after second loop words = " + words);
28     }
29 }
```

If we want to write the loops in a more intuitive manner, we can run them backwards. In other words, the loops we have written go from left to right, from the beginning of the list to the end of the list. We could instead go right to left, from the end of the list to the beginning of the list. By going backwards, we ensure that any changes we are making occur in parts of the list that we have already visited.

For example, we found that this loop did not work properly even though it seemed like the intuitive approach:

```
// doesn't work properly
for (int i = 0; i < words.size(); i++) {
    words.add(i, "*");
}
```

But if we turn this loop around and have it go backwards rather than going forwards, then it does work properly:

```
// works properly because loop goes backwards
for (int i = words.size() - 1; i >= 0; i--) {
    words.add(i, "*");
}
```

The problem we had with the original code was that we were inserting a value into the list and then moving our index variable onto that spot in the list. By going backwards, the changes that we make happen in parts of the list that we have already processed.

Similarly, we wanted to write the second loop this way:

```
// doesn't work properly
for (int i = 0; i < words.size(); i += 2) {
    words.remove(i);
}
```

Again, the problem is that we are changing a part of the list that we are about to process. We can keep the overall structure by running the loop backwards:

```
// works properly because loop goes backwards
for (int i = words.size() - 1; i >= 0; i -= 2) {
    words.remove(i);
}
```

The for-each Loop

We saw in Chapter 7 that we can use the for-each loop to iterate over the elements of an array. We can do the same with the ArrayList. For example, we saw that the following code could be used to add up the lengths of the Strings stored in an ArrayList<String> called list:

```
int sum = 0;
for (int i = 0; i < list.size(); i++) {
    String s = list.get(i);
    sum += s.length();
}
System.out.println("Total of lengths = " + sum);
```

We can simplify this code with the for-each loop. Remember that its syntax is as follows:

```
for (<type> <name> : <structure>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

The loop above can be rewritten as follows:

```
int sum = 0;
for (String s : list) {
    sum += s.length();
}
System.out.println("Total of lengths = " + sum);
```

Remember that you can think of this loop as saying, "For each String s contained in list...". Because the for-each loop has such a simple syntax, you should use it whenever you find yourself wanting to sequentially process each value stored in a list.

You will find, however, that the for-each loop is not appropriate for more complex list problems. For example, there is no simple way to skip around in the list using the for-each loop. You must process the values in sequence from first to last. You also are not allowed to modify the list while you are iterating over it.

Consider, for example, the following sample code:

```
for (String s : words) {
    System.out.println(s);
    words.remove(0);
}
```

This code prints each String from the list and then attempts to remove the value at the front of the list. When you execute this code, the program halts with a ConcurrentModificationException. Java is letting you know that you are not allowed to iterate over the list and to modify the list at the same time (concurrently). Because of this limitation, neither of the problems from the previous section could be solved using the for-each loop.

Wrapper Classes

So far all of the ArrayList examples we have looked at have involved an ArrayList of String objects. What if we wanted to form a list of integers? Given that the ArrayList<E> is a generic class, you'd think that Java would allow you to define an ArrayList<int>, but that is not the case. The E in ArrayList<E> can be filled in with any object or reference type (i.e., the name of a class). The primitive types like int, double, char and boolean cannot be used as the type parameter for an ArrayList.

Instead, Java defines a series of *wrapper classes* that allow you to store primitive data as objects.

Wrapper Class

A class that "wraps" primitive data as an object.

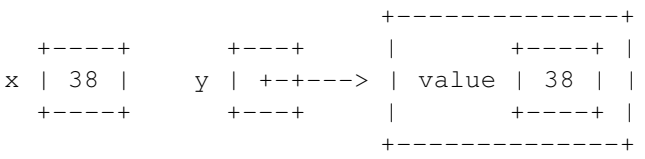
To understand the role of a wrapper class, think about why we so often put candy in a wrapper. Pieces of candy can be sticky and inconvenient to handle directly, so we put them inside a wrapper that makes them more convenient to handle. When we want the actual candy, we open up the wrapper to get the candy out. The Java wrapper classes fill a similar role.

Consider, for example, simple integers. We have seen that they are of type int, which is a primitive type. Primitive types are not objects, which means that we can't use values of type int in an object context. So we wrap up each int into an object of type Integer. Integer objects are very simple. They have just one data field: an int value. When we construct an Integer, we pass an int value to be wrapped. And when we want to get the int back, we call a method intValue that returns the int.

To understand the distinction, consider the following variable declarations:

```
int x = 38;
Integer y = new Integer(38);
```

This code leads to the following situation in memory:



We have seen that primitive data is stored directly (the variable x stores the actual value 38) while objects are stored as references (the variable y stores a reference to an object that contains 38).

If we later want to get the 38 out of the object (to unwrap it and remove the candy inside), we call the method intValue:

```
int number = y.intValue();
```

The wrapper classes are of particular interest for this chapter because if you want to have an ArrayList<E>, then the E needs to be a reference type. So we can't form an ArrayList<int>, but we can form an ArrayList<Integer>. For example, we can write the following code that adds several integer values to a list and adds them up:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(13));
list.add(new Integer(47));
list.add(new Integer(15));
list.add(new Integer(9));
int sum = 0;
for (Integer n : list) {
    sum += n.intValue();
}
System.out.println("list = " + list);
System.out.println("sum  = " + sum);
```

This code produces the following output:

```
list = [13, 47, 15, 9]
sum  = 84
```

Java provides a mechanism for simplifying code that involves the use of wrapper classes. In particular, Java will convert between Integer values and int values when your intent seems clear. For example, given the declaration of the variable list as an ArrayList<Integer>, we can add a series of Integer objects to the list, but we can also simply add int values, as in:

```
list.add(13);
```

In this line of code Java sees that you are adding an int to a structure that is expecting an Integer. But Java also knows the relationship between int and Integer (that each Integer is simply an int wrapped up as an object). So in this case Java will automatically convert the int value into a corresponding Integer object. This process is known as *boxing*.

Boxing

An automatic conversion from primitive data to a wrapped object of appropriate type (e.g., an int boxed to form an Integer).

Similarly, we don't have to call the intValue method to unwrap the object. The body of the for loop can be rewritten as follows:

```
sum += n;
```

In this case we have a variable of type int on the left-hand side and a variable of type Integer on the right-hand side. Normally these variables would be incompatible except for the fact that Java knows about the relationship between int and Integer. So in this case, Java will call the intValue method for you to unwrap the object. This process is known as *unboxing*.

Unboxing

An automatic conversion from a wrapped object to its corresponding primitive data (e.g., an Integer unboxed to yield an int).

We can even rewrite our for-each loop to use a variable of type int even though the ArrayList stores values of type Integer. Because of automatic unboxing, Java will perform the appropriate conversions for us. So the code fragment above can be rewritten as follows:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(13);
list.add(47);
list.add(15);
list.add(9);
int sum = 0;
for (int n : list) {
    sum += n;
}
System.out.println("list = " + list);
System.out.println("sum  = " + sum);
```

Because of boxing and unboxing, you will find that the only place you generally need to use the wrapper class is in defining a type like ArrayList<Integer>. You can't define it to be of type ArrayList<int>, but even though it is of type ArrayList<Integer>, you can manipulate it as if it is of type ArrayList<int>.

Below is a list of the major primitive types and their corresponding wrapper classes.

Common Wrapper Classes

Primitive Type	Wrapper Class
int	Integer
double	Double
char	Character
boolean	Boolean

Did you Know: Controversy Over Boxing and Unboxing

Not all software developers are happy with Sun's decision to add boxing and unboxing to the language. The fact that an ArrayList<Integer> can be manipulated almost as if it were an ArrayList<int> can simplify code and everyone agrees that simplification is good. But the

disagreement comes from the fact that it is *almost* like an ArrayList<int>. The argument is that "almost" isn't good enough. The fact that it comes close means that you are likely to use it and eventually come to count on it. That can prove disastrous when "almost" isn't "always."

For example, suppose that someone told you to use a device that is almost like a potholder. In most cases, it will protect your hand from heat. So you start using it and you might be nervous at first, but then you find that it seems to work just fine. And then one day you're surprised to find that a small area isn't behaving like a potholder and you get burned. You can think of similar scenarios with aircraft landing gear that almost works or vests that are almost bullet-proof.

For example, consider the following code:

```
int n = 42;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(n);
list.add(n);
if (list.get(0) == list.get(1)) {
    System.out.println("equal");
} else {
    System.out.println("unequal");
}
```

It's difficult to know exactly what this code will do. If we think of the ArrayList<Integer> as being "almost" like an ArrayList<int>, we would be inclined to think that the code would print the message that the two values are equal. The code does in fact print that message.

But remember that testing for object equality is not as simple as testing for equality of primitive data. Two Strings might store the same text but not be the same object, which is why we call the equals method to compare Strings. The same applies here. The two list elements might store the same int but not be the same object. The code above happens to work, but depending upon the value of n, it isn't guaranteed to work. If we change the assignment of n to 420 instead of 42, the code prints the message that the values are unequal.

The Java Language Specification guarantees that this code will work for any value of n between -128 and 127, but provides no guarantee as to how the code will behave for other values of n. It could print either message. And we have no guarantee of how this might change from one implementation of Java to another. In might be that in the next release of Java, the code will print "equals" for 420 but not for a value like 420000.

Some have argued that because boxing and unboxing cover up what is happening underneath, that it is better not to use them at all. Boxing and unboxing don't necessarily simplify anything if they work only "sometimes" because you have to be able to understand the cases where they don't work.

10.2 The Comparable Interface

We saw in Chapter 7 that arrays can be sorted by calling the method Arrays.sort that is part of the java.util package. There is a similar method that can be used to sort an ArrayList that is called Collections.sort. It is also part of the java.util package. The following short program demonstrates how to use Collections.sort.

```
1  // Short program that constructs an ArrayList of Strings and sorts it.
2
3  import java.util.*;
4
5  public class SortExample {
6      public static void main(String[] args) {
7          ArrayList<String> words = new ArrayList<String>();
8          words.add("four");
9          words.add("score");
10         words.add("and");
11         words.add("seven");
12         words.add("years");
13         words.add("ago");
14
15         // show list before and after sorting
16         System.out.println("before sort, words = " + words);
17         Collections.sort(words);
18         System.out.println("after sort, words = " + words);
19     }
20 }
```

This program produces the following output:

```
before sort, words = [four, score, and, seven, years, ago]
after sort, words = [ago, and, four, score, seven, years]
```

If we try to do something similar with an ArrayList<Point>, we will find that the program does not compile. Why can we sort a list of String objects but not a list of Point objects? The answer is that the String class implements an interface known as the Comparable interface while the Point class does not. In this section we will explore the details of the Comparable interface and learn how to write classes that implement it.

Natural Ordering and compareTo

We are familiar with many kinds of data that can be sorted. For example, we are used to putting numbers into increasing order or alphabetizing a list of names. We describe types that can be sorted as having a *natural ordering* of values. To have such an ordering of values, a type needs a well-defined *comparison function* that indicates the relationship between any pair of values.

Comparison Function

A well-defined procedure for deciding, given a pair of values, the relative order of the two values (less-than, equal or greater-than).

Natural Ordering

The order imposed on a type by its comparison function.

Not all types have a natural ordering because not all types have a comparison function. For example, we have been exploring in this chapter how to construct a variety of ArrayList objects. How would you compare two ArrayList objects to determine that one is less than another? What would it mean for one ArrayList to be less than another? You might decide to use the length of the list to determine which one is less, but then what do you do with two ArrayList objects of equal

length that store different values? You wouldn't want to describe them as "equal". There is no natural ordering for ArrayList objects because we have no comparison function to figure out what to do with each such pair of values.

Java has a convention for indicating the natural ordering of a type. Any type that has such an ordering should implement the Comparable interface.

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

This interface provides a second example of a generic type in Java. In the case of ArrayList Sun decided to use the letter "E" which is short for "Element". Here Sun decided to use the letter "T" which is short for "Type".

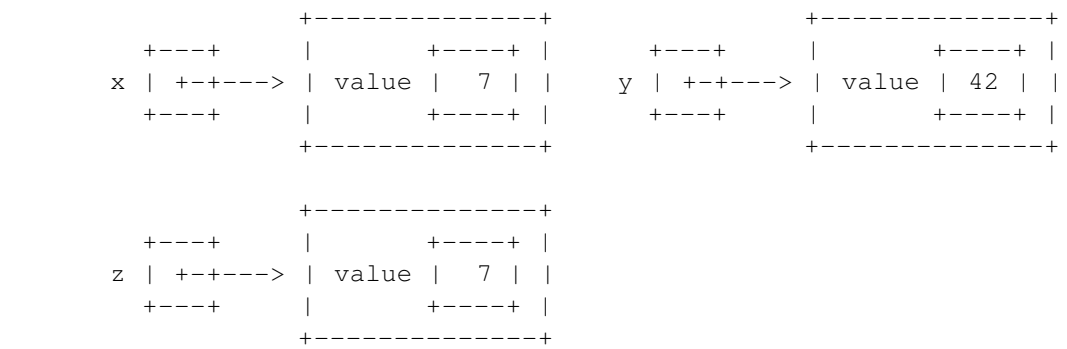
The compareTo method is the comparison function for the type. We can't use a boolean return type because there are three possible answers: less-than, equal or greater. The convention for compareTo is that an object should return:

- a negative number to indicate a less-than relationship
- 0 to indicate equality
- a positive number to indicate a greater-than relationship

Let's look at a few examples to understand this better. We have seen that Java has Integer objects that serve as wrappers for individual int values. We know how to compare int values to determine their relative order so it is not surprising that the Integer class implements the Comparable interface. Consider the following short program:

```
 1  public class CompareExample1 {
 2      public static void main(String[] args) {
 3          Integer x = 7;
 4          Integer y = 42;
 5          Integer z = 7;
 6          System.out.println(x.compareTo(y));
 7          System.out.println(x.compareTo(z));
 8          System.out.println(y.compareTo(x));
 9      }
10  }
```

This program begins by constructing three Integer objects:



Then it includes a series of println statements that report the results of various pairwise comparisons. In the first println, we ask x to compare itself to y, which involves comparing the int value 7 to the int value 42. This pair has a less-than relationship (x is less than y), so the method call returns a negative integer. In the second println, we ask x to compare itself to z, which involves comparing one occurrence of the int value 7 with another occurrence of the int value 7. This second pair has an equality relationship (x equals z), so the method call returns 0. In the final println we ask z to compare itself to x, which involves comparing the int value 42 to the int value 7. This final pair has a greater-than relationship (z is greater than x), so the method call returns a positive integer.

Below is the actual output of the program:

```
-1
0
1
```

The values -1, 0 and 1 are the standard values to return, but the compareTo method is not required to return these specific values. For example, consider a similar program that compares String values:

```
 1  public class CompareExample2 {
 2      public static void main(String[] args) {
 3          String x = "hello";
 4          String y = "world";
 5          String z = "hello";
 6          System.out.println(x.compareTo(y));
 7          System.out.println(x.compareTo(z));
 8          System.out.println(y.compareTo(x));
 9      }
10  }
```

We have similar relationships in this program because x is less than y ("hello" is less than "world"), x is equal to z (the two occurrences of "hello" are equal) and z is greater than x ("world" is greater than "hello"). But the output produced is slightly different than in the Integer example:

```
-15
0
15
```

Instead of -1 and 1, we get -15 and 15. So while the values -1 and 1 are often returned by a comparison function, that won't always be the case. The important thing to remember is that "less-than" relationships are indicated by a negative number and "greater-than" relationships are indicated by a positive number.

Keep in mind that the relationship operators that we've been using since Chapter 4 have a different syntax. For example, we have seen that if you have two variables x and y that are of type int or double, you can compare them using operators like "<" and ">", as in:

```
int x = 7;
int y = 42;
if (x < y) {
    System.out.println("x less than y");
}
```

Even though the Integer class and String class implement the Comparable interface, you can't use the relational operator "<" to compare them.

```
Integer x = 7;
Integer y = 42;
if (x < y) {  // illegal--can't compare objects this way
    System.out.println("x less than y");
}
```

Instead, we call the compareTo method, as in:

```
Integer x = 7;
Integer y = 42;
if (x.compareTo(y) < 0) {
    System.out.println("x less than y");
}
```

We can use a relational operator in this context because the compareTo method returns an int. Notice that we don't use the specific value of -1 for compareTo because we are only guaranteed to get a negative value for a less-than relationship. The if statement above works equally well for the String class:

```
String x = "hello";
String y = "world";
if (x.compareTo(y) < 0) {
    System.out.println("x less than y");
}
```

Below is a table that summarizes the standard

Comparing Values Summary

Relationship	Primitive Data (int, double, etc)	Objects (Integer, String, etc)
less-than	`if (x < y) { ... }`	`if (x.compareTo(y) < 0) { ... }`
less-than or equal	`if (x <= y) { ... }`	`if (x.compareTo(y) <= 0) { ... }`
equal	`if (x == y) { ... }`	`if (x.compareTo(y) == 0) { ... }`
not equal	`if (x != y) { ... }`	`if (x.compareTo(y) != 0) { ... }`
greater-than	`if (x > y) { ... }`	`if (x.compareTo(y) > 0) { ... }`
greater-than or equal	`if (x > y) { ... }`	`if (x.compareTo(y) >= 0) { ... }`

Implementing Comparable

Let's explore how to implement the Comparable interface. To keep things relatively simple, let's explore a class that would keep track of a calendar date. The idea is to keep track of a particular month and day without keeping track of a year. For example, the United States celebrates its independence on July 4th each year. Similarly an organization might want a birthday list that doesn't include information about how old people are.

We can implement this as a class with two data fields to store the month and day:

```
public class CalendarDate {
    private int month;
    private int day;

    public CalendarDate(int month, int day) {
    this.month = month;
    this.day = day;
    }

    // other methods
}
```

Remember that to implement an interface, you include an extra notation in the class header. Implementing the Comparable interface is a little more challenging because it is a generic interface (Comparable<T>). So we can't simply say:

```
public class CalendarDate implements Comparable {  // not correct
    ...
}
```

We have to replace the <T> in Comparable<T>. Whenever you implement Comparable, you will be comparing pairs of values from the same class. So a class called CalendarDate should implement Comparable<CalendarDate>. If you look at the header for the Integer class, you will find that it implements Comparable<Integer>. And the String class implements Comparable<String>. So we need to change the header to the following:

```
public class CalendarDate implements Comparable<CalendarDate> {
    ...
}
```

Claiming to implement the interface is not enough. We also have to include appropriate methods. In this case, the Comparable interface has just a single method for us to include:

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

Because we are using CalendarDate in place of T, we need to write a compareTo method that takes a parameter of type CalendarDate:

```
public int compareTo(CalendarDate other) {
   ...
}
```

Now we have to figure out how to compare two dates. Each CalendarDate object will have data fields storing month and day. With calendars, the month takes precedence over the day. If we want to compare January 31 (1/31) with April 5 (4/5), we don't care that 5 comes before 31, we care more about the fact that January comes before April. So as a first attempt, we could write the method as follows:

```
public int compareTo(CalendarDate other) {
    if (this.month < other.month) {
        return -1;
    } else if (this.month == other.month) {
        return 0;
    } else { // this.month > other.month
        return 1;
    }
}
```

There are two problems with this approach. First, this code uses a nested if/else to return the standard values of -1, 0 and 1. We have a simpler option available to us. We can simply return the difference between this.month and other.month because it will be negative when this.month is less than other.month, it will be 0 when they are equal and it will be positive when this.month is greater than other.month. So we can simplify the code as follows:

```
public int compareTo(CalendarDate other) {
    return this.month - other.month;
}
```

This version returns slightly different values than the earlier version, but it satisfies the contract of the Comparable interface just as well as the other version. It is a good idea to keep things simple when you can, so we would tend to use this version. But the code still has a problem.

While it's true that months are more important than days in a calendar, the days can be important. Consider, for example, April 1 (4/1) versus April 5 (4/5). The current version of compareTo would subtract the months and return a value of 0, indicating that these two dates are equal. The dates aren't equal. In particular, April 1 comes before April 5.

If you think about how we put dates into order, you'll realize that the day of the month becomes important only when the months are equal. So if the months differ, then we use the months to determine order. Otherwise (when the months are equal), we use the day of the month to determine order. This is a common ordering principle that you will find in many tasks. We can implement this strategy as follows:

```
public int compareTo(CalendarDate other) {
    if (this.month != other.month) {
        return this.month - other.month;
    } else {
        return this.day - other.day;
    }
}
```

It might seem that this code never returns 0, but it does. Suppose that we have two CalendarDate objects that both store April 5 (4/5). We find that the months are equal, so we return the difference between the dates. That difference is 0, so we return 0, which correctly indicates that the two dates are equal.

Below is a complete class with the compareTo method, two accessor methods and a toString method:

```
1  // The CalendarDate class stores information about a single calendar date
2  // (month and day but no year).
3
4  public class CalendarDate implements Comparable<CalendarDate> {
5      private int month;
6      private int day;
7
8      public CalendarDate(int month, int day) {
9          this.month = month;
10         this.day = day;
11     }
12
13     // Compares this calendar date to another date.
14     // Dates are compared by month and then by day.
15     public int compareTo(CalendarDate other) {
16         if (this.month != other.month) {
17             return this.month - other.month;
18         } else {
19             return this.day - other.day;
20         }
21     }
22
23     public int getMonth() {
24         return this.month;
25     }
26
27     public int getDay() {
28         return this.day;
29     }
30
31     public String toString() {
32         return this.month + "/" + this.day;
33     }
34 }
```

One of the major benefits of implementing the Comparable interface is that you can use built-in utilities like Collections.sort. We saw that we could use Collections.sort to sort an ArrayList<String> but not to sort an ArrayList<Point>. Below is a short program demonstrating the fact that we can use Collections.sort for an ArrayList<CalendarDate>. That works because the CalendarDate class implements the Comparable interface.

```
 1  // Short program that creates a list of the birthdays of the first 5
 2  // US Presidents and that puts them into sorted order.
 3
 4  import java.util.*;
 5
 6  public class CalendarDateTest {
 7      public static void main(String[] args) {
 8          ArrayList<CalendarDate> dates = new ArrayList<CalendarDate>();
 9          dates.add(new CalendarDate(2, 22));   // Washington
10          dates.add(new CalendarDate(10, 30));  // Adams
11          dates.add(new CalendarDate(4, 13));   // Jefferson
12          dates.add(new CalendarDate(3, 16));   // Madison
13          dates.add(new CalendarDate(4, 28));   // Monroe
14
15          System.out.println("birthdays = " + dates);
16          Collections.sort(dates);
17          System.out.println("birthdays = " + dates);
18      }
19  }
```

This program produces the following output:

```
birthdays = [2/22, 10/30, 4/13, 3/16, 4/28]
birthdays = [2/22, 3/16, 4/13, 4/28, 10/30]
```

Notice that the dates appear in increasing calendar order after the call on Collections.sort.

Did you Know: Why not -1, 0 and 1 for compareTo?

As we have seen, some types like Integer return the values of -1, 0 and 1 when you call compareTo. These are the *canonical* values for compareTo to return because they correspond to a function in Mathematics known as the *signum* function (sometimes abbreviated "sgn"). But other types like String do not return the standard values of -1, 0 and 1. You might wonder why Sun didn't require that all classes return -1, 0 and 1 when you call compareTo.

One answer to this question is that Java doesn't have a convenient ternary type. For any binary decision we can use boolean as the return type. But what type do we turn to if we want to return exactly one of three different values? We don't have a convenient predefined type with just three values. So it's more honest in some sense to use Sun's rule that any negative number will do and any positive number will do. Suppose that Sun said that compareTo should return just -1, 0 and 1. Then what should happen when someone writes a compareTo that returns something else? Ideally any code calling that compareTo would throw an exception when it gets an illegal return value, but that would require a lot of error checking code. By saying that all negatives will be interpreted one way, all positives will be interpreted a second way and 0 will be interpreted a third way, Sun is providing a complete definition for all values of type int which makes the compareTo method easier for programmers to work with.

A second reason for having compareTo behave this way is that many comparison tasks can be easily expressed directly in this way. We saw that it simplified our CalendarDate code to return either the difference in the months (when the months were unequal) or the difference in the days (when the months were equal). This pattern occurs in many places. For example, the String class uses *lexicographic* order (also called "dictionary" or "alphabetic" order). To determine the relationship between two Strings, you scan through them trying to find the first pair of letters that differ. For example, if you were comparing "nattering" and "nabobs", you'd find that the first

pair of characters that differ is the third pair ("nat..." versus "nab..."). If you find such a pair, you can return the difference between the character values ('t' - 'b'). If you don't find such a pair, then you return the difference between the lengths so that "nattering" will be considered greater than "nat" based on length.

In other words, the compareTo behavior for the String class can be described with the following pseudocode:

```
search for a pair of characters in corresponding positions that differ.
if (such a pair exists) {
    return this.char - other.char;
} else {
    return this.length - other.length;
}
```

Notice that this approach returns 0 in just the right case when there are no character pairs that differ and when the strings have the same length. Having the flexibility to return any negative integer for "less than" and any positive integer for "greater than" makes it easier to implement this approach.

The final reason to have compareTo behave this way is efficiency. By having a less strict rule, Sun allows programmers to write faster compareTo methods. For example, one of the most frequently called methods is the compareTo method in the String class. All sorts of data comparisons are built on String comparisons and performing a task like sorting thousands of records is going to lead to thousands of calls on the String class compareTo method. As a result, we want to make sure that the method runs quickly, so we wouldn't want to unnecessarily complicate the code by requiring that it always return -1, 0 or 1.

10.3 Case Study: Vocabulary Comparison

In this section we will use ArrayLists to solve a complex problem. We are going to read two different text files and compare their vocabulary. Researchers in the Humanities often perform such comparisons to answer questions like, "Did Christopher Marlowe actually write Shakespeare's plays?" In particular, we will determine the set of words used in each file and compute how much overlap they have.

As we have done with most of the case studies, we will develop the program in stages:

1. The first version will read the two files and report the unique words in each. We will use short testing files for this stage.
2. The second version will also compute the overlap between the two. We will continue to use short testing files for this stage.
3. We will read from large text files and will include some analysis of the results.

Version 1: Compute Vocabulary

The program we are writing will only be interesting when we use large input files to compare, but while we are developing the program, it is best to use short input files where we can easily check whether we are getting the right answer. By using short input files we also don't have to worry about execution time. When you use a large input file and the program takes a long time to execute,

it is difficult to know whether the program will ever finish executing. By developing the program with short input files, we know that it should never take a long time to execute. So if we accidentally introduce an infinite loop into our program, we'll know right away that the problem is with our code, not with the fact that we have a lot of data to process.

For our purposes, let's use the first two stanzas of a popular children's song as our input files. So we can create a file called test1.txt that contains the following:The wheels on the bus go round and round round and round round and round. The wheels on the bus go round and round all through the town.

And we can create a file called test2.txt that contains the following:

```
The wipers on the bus go Swish, swish, swish,
Swish, swish, swish,
Swish, swish, swish.
The wipers on the bus go Swish, swish, swish,
all through the town.
```

We need to open each of these files with a Scanner, so our main method will begin with:

```
Scanner input1 = new Scanner(new File("test1.txt"));
Scanner input2 = new Scanner(new File("test2.txt"));
```

Then we want to compute the unique vocabulary contained in each file. We can store this in an ArrayList<String>. The operation will be the same for each, so it makes sense to write a single method that we call twice. The method should take the Scanner as a parameter and it should convert that into an ArrayList<String> that contains the vocabulary. So after opening the files, we can execute the following code:

```
ArrayList<String> list1 = getWords(input1);
ArrayList<String> list2 = getWords(input2);
```

This version is meant to be fairly simple, so after we have computed the vocabulary for each file, we can simply report it:

```
System.out.println("list1 = " + list1);
System.out.println("list2 = " + list2);
```

So the difficult work for this version of the program reduces to writing the getWords method. It should read all of the words from the Scanner, building up an ArrayList<String> that contains those words. We can do this fairly easily with the ArrayList contains method. For our purposes, we don't care about capitalization, so we can convert each word to lowercase before we add it to the list. So we can build up the list with the following code:

```
ArrayList<String> words = new ArrayList<String>();
while (input.hasNext()) {
    String next = input.next().toLowerCase();
    if (!words.contains(next)) {
        words.add(next);
    }
}
```

Putting these pieces together, we get the following complete program:

```
 1  import java.util.*;
 2  import java.io.*;
 3
 4  public class Vocabulary1A {
 5      public static void main(String[] args) throws FileNotFoundException {
 6          Scanner input1 = new Scanner(new File("test1.txt"));
 7          Scanner input2 = new Scanner(new File("test2.txt"));
 8
 9          ArrayList<String> list1 = getWords(input1);
10          ArrayList<String> list2 = getWords(input2);
11
12          System.out.println("list1 = " + list1);
13          System.out.println("list2 = " + list2);
14      }
15
16      public static ArrayList<String> getWords(Scanner input) {
17          ArrayList<String> words = new ArrayList<String>();
18          while (input.hasNext()) {
19              String next = input.next().toLowerCase();
20              if (!words.contains(next)) {
21                  words.add(next);
22              }
23          }
24          return words;
25      }
26  }
```

This program behaves fairly well. When we execute it, we get the following output:

```
list1 = [the, wheels, on, bus, go, round, and, round., all, through, town.]
list2 = [the, wipers, on, bus, go, swish,, swish., all, through, town.]
```

The original input files each have 28 words in them. We have reduced the first file to 11 unique words and the second to 10 unique words. The program is correctly ignoring differences in case, but it isn't ignoring differences in punctuation. For example, it considers "round" and "round." to be different words (one with a period, one without). If we had more time to invest, we could explore ways to configure the Scanner so that it would ignore such differences, but for our purposes this will be good enough.

So this approach works, but it is not likely to work fast enough for large text files. As the list grows, the contains method is going to have to look at more and more values. We will also find it challenging to compare the two lists in this form.

Many problems like this one are easier to solve if we can put the list into sorted order. Because we are using an ArrayList<String>, we can call Collections.sort to put the list into sorted order. So we can rewrite the beginning of the method to store all words into the ArrayList and sort the list after filling it up:

```
while (input.hasNext()) {
    String next = input.next().toLowerCase();
    words.add(next);
}
Collections.sort(words);
```

Once the list has been sorted, all of the duplicates of a word will be grouped together. Because the duplicates will be next to each other, we can compare adjacent elements and remove any duplicates that we encounter. Below is a pseudocode description of this approach:

```
for (each i) {
    if (value at i equals value at i+1) {
        remove value at i+1.
    }
}
```

While this approach can work, it turns out to be too slow as well. Remember that a call on remove requires shifting values. If the list has thousands of words in it, then each of these calls on remove is likely to be time-consuming.

Instead of removing duplicates from this list, we can instead build up a new list that has the unique words. If we add the words in alphabetical order, we can always add at the end of the new list and adding at the end of a list is very fast.

In the first approach we were looking for duplicates to remove. In this second approach, we need to look for unique words. The simplest way to do this is to look for transitions between words. For example, if we have five occurrences of one word followed by ten occurrences of another word, then most of the pairs of adjacent words will be equal to each other. But in the middle of those equal pairs there will be a pair that are not equal when we make the transition from the first word to the second word. Whenever we see such a transition, we know that we are seeing a new word that should be added to our new list.

Looking for transitions leads to a classic fencepost problem. For example, if there are 10 unique words, there will be 9 transitions. We can solve the fencepost problem by adding the first word before the loop begins. Then we can look for words that are not equal to the words that come before them and add them to the list. Expressed as pseudocode, we get the following:

```
construct a new empty list.
add first word to new list.
for (each i) {
    if (value at i does not equal value at i-1) {
        add value at i.
    }
}
```

This can be converted into actual code fairly directly, but we have to be careful to start i at 1 rather than 0 because in the loop we compare each word as the one that comes before it and the first word has nothing before it. We also have to be careful to call the ArrayList get method to obtain individual values and to use the equals method to compare Strings for equality:

```
ArrayList<String> result = new ArrayList<String>();
result.add(words.get(0));
for (int i = 1; i < words.size(); i++) {
    if (!words.get(i).equals(words.get(i - 1))) {
        result.add(words.get(i));
    }
}
```

There is still one minor problem with this code. If the input file is empty, then there won't be a first word to add to the new list. So we need an extra if to make sure that we don't try to add values to the new list if the first list is empty.

Putting all of these changes together, we get the following program.

```
 1  import java.util.*;
 2  import java.io.*;
 3
 4  public class Vocabulary1B {
 5      public static void main(String[] args) throws FileNotFoundException {
 6          Scanner input1 = new Scanner(new File("test1.txt"));
 7          Scanner input2 = new Scanner(new File("test2.txt"));
 8
 9          ArrayList<String> list1 = getWords(input1);
10          ArrayList<String> list2 = getWords(input2);
11
12          System.out.println("list1 = " + list1);
13          System.out.println("list2 = " + list2);
14      }
15
16      public static ArrayList<String> getWords(Scanner input) {
17          // read all words and sort
18          ArrayList<String> words = new ArrayList<String>();
19          while (input.hasNext()) {
20              String next = input.next().toLowerCase();
21              words.add(next);
22          }
23          Collections.sort(words);
24
25          // add unique words to new list and return
26          ArrayList<String> result = new ArrayList<String>();
27          if (words.size() > 0) {
28              result.add(words.get(0));
29              for (int i = 1; i < words.size(); i++) {
30                  if (!words.get(i).equals(words.get(i - 1))) {
31                      result.add(words.get(i));
32                  }
33              }
34          }
35          return result;
36      }
37  }
```

The program produces the following output:

```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
```

Version 2: Compute Overlap

The first version of the program produces two sorted ArrayLists that each have a unique set of words in them. For the second version, we want to compute the overlap between the two words and report it. This operation will be complex enough that it deserves to be in its own method. So we can add the following line of code to the main method right after the two word lists are constructed:

```
ArrayList<String> overlap = getOverlap(list1, list2);
```

Thus, the primary task for this second version is to implement the getOverlap method. Look closely at the two lists of words from the first version:

```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
```

People are pretty good at finding matches, so you can probably see exactly what words overlap. Both lists begin with "all", so that is part of the overlap. Skipping past the word "and" in the first list, we find the next match is for the word "bus". Then we have another two matches with the words "go" and "on". And then we have several words in a row in both lists that don't match followed eventually by the match with the word "the". And we find one final math with the word "through". So the complete set of matches is as follows:

```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
          |          /   /   /                  /       /
          |        /   /   /                   /      /
          |      /   /   /                     /     /
list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
```

We want to design an algorithm that parallels what we do when we look for such matches. Imagine putting a finger from your left hand on the first word in the first list and putting a finger from your right hand on the first word in the second list to keep track of where you are in each list. Then we will repeatedly compare the words you are pointing at. Depending upon how they compare, we will move one or both fingers forward.

We start with the left finger on the word "all" in the first list and the right finger on the word "all" in the second list.

```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
          ^

list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
          ^
```

The words you are pointing at match, so you add that word to the overlap and move both fingers forward:

```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
               ^

list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
               ^
```

These words don't match. So what do you do? It turns out that the word "bus" in list2 is going to match a word in list1. So how do you know to move the left finger forward? We know that because we are pointing at the word "and" from the first list and the word "bus" from the second list. Because the list is sorted and because the word "and" comes before the word "bus", we know there can't be a match for the word "and" in the second list. Every word that comes after "bus" in the second list is going to be alphabetically greater than "bus", so the word "and" just can't be there. So we can move the left finger forward to skip the word "and":

```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
                    ^

list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
                 ^
```

This gets us to the second match and the algorithm proceeds. In general, we will find ourselves in one of three situations when we compare the current word in list1 with the current word in list2:

- The words might be equal, in which case we've found a match that should be included in the overlap and we should then skip the word in each list.
- The word from the first list might be alphabetically less, in which case we can skip it because it can't match anything in the second list.
- The word from the second list might be alphabetically less, in which case we can skip it because it can't match anything in the first list.

Thus, the basic approach we want to use can be described with the following pseudocode:

```
if (word from list1 equals word from list2) {
    record match.
    skip past word in each list.
} else if (word from list1 < word from list2) {
    skip past word in list1.
} else {
    skip past word in list2.
}
```

We can refine this pseudocode by introducing two index variables and putting this code inside of a loop:

```
int index1 = 0;
int index2 = 0;
while (more values to compare) {
    if (list1.get(index1) equals list2.get(index2)) {
        record match.
        index1++;
        index2++;
    } else if (list.get(index1) < list.get(index2)) {
        index1++;
    } else {
        index2++;
    }
}
```

This is now fairly close to actual code. First, we have to figure out an appropriate loop test. We start the two index variables at 0 and increment one or both each time through the loop. Eventually we'll run out of values in one or both lists. When that happens, there won't be any more matches to find. So we want to continue in the while loop as long as the two index variables haven't reached the end of the list. We also have to figure out how to compare the two words. Remember that the String class implements the Comparable interface, which means it has a compareTo method. Finally, we have to construct an ArrayList to store the overlap and we have to return it after the loop.

Thus, we can turn this into the following actual code:

```
ArrayList<String> result = new ArrayList<String>();
int index1 = 0;
int index2 = 0;
while (index1 < list1.size() && index2 < list2.size()) {
    int comparison = list1.get(index1).compareTo(list2.get(index2));
    if (comparison == 0) {
        result.add(list1.get(index1));
        index1++;
        index2++;
    } else if (comparison < 0) {
        index1++;
    } else {  // comparison > 0
        index2++;
    }
}
return result;
```

Turning this into a method and modifying main to call this method and to report the overlap, we end up with the following new version of the program:

```
 1  import java.util.*;
 2  import java.io.*;
 3
 4  public class Vocabulary2 {
 5      public static void main(String[] args) throws FileNotFoundException {
 6          Scanner input1 = new Scanner(new File("test1.txt"));
 7          Scanner input2 = new Scanner(new File("test2.txt"));
 8
 9          ArrayList<String> list1 = getWords(input1);
10          ArrayList<String> list2 = getWords(input2);
11          ArrayList<String> overlap = getOverlap(list1, list2);
12
13          System.out.println("list1 = " + list1);
14          System.out.println("list2 = " + list2);
15          System.out.println("overlap = " + overlap);
16      }
17
18      public static ArrayList<String> getWords(Scanner input) {
19          // read all words and sort
20          ArrayList<String> words = new ArrayList<String>();
21          while (input.hasNext()) {
22              String next = input.next().toLowerCase();
23              words.add(next);
24          }
25          Collections.sort(words);
26
27          // add unique words to new list and return
28          ArrayList<String> result = new ArrayList<String>();
29          if (words.size() > 0) {
30              result.add(words.get(0));
31              for (int i = 1; i < words.size(); i++) {
32                  if (!words.get(i).equals(words.get(i - 1))) {
33                      result.add(words.get(i));
34                  }
35              }
36          }
37          return result;
38      }
39
```

```
40      public static ArrayList<String> getOverlap(ArrayList<String> list1,
41                                                 ArrayList<String> list2) {
42          ArrayList<String> result = new ArrayList<String>();
43          int index1 = 0;
44          int index2 = 0;
45          while (index1 < list1.size() && index2 < list2.size()) {
46              int comparison = list1.get(index1).compareTo(list2.get(index2));
47              if (comparison == 0) {
48                  result.add(list1.get(index1));
49                  index1++;
50                  index2++;
51              } else if (comparison < 0) {
52                  index1++;
53              } else {  // comparison > 0
54                  index2++;
55              }
56          }
57          return result;
58      }
59  }
```

It produces the following output:

```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
overlap = [all, bus, go, on, the, through, town.]
```

Version 3: Complete Program

Our program now correctly builds a vocabulary list for each of two files and computes the overlap between them. The last version of the program printed the three lists of words, but that won't be very convenient for large text files. Large text files have thousands of different words. We can instead report overall statistics including the number of words in each list, the number of words of overlap and the percent of overlap.

The program also should have at least a brief introduction to explain what it does and we can write it so that it prompts for file names rather than using hard-coded file names. Below is the complete program:

```
1  // This program reads two text files and compares the vocabulary used in each.
2
3  import java.util.*;
4  import java.io.*;
5
6  public class Vocabulary3 {
7      public static void main(String[] args) throws FileNotFoundException {
8          Scanner console = new Scanner(System.in);
9          giveIntro();
10
11         System.out.print("file #1 name? ");
12         Scanner input1 = new Scanner(new File(console.nextLine()));
13         System.out.print("file #2 name? ");
14         Scanner input2 = new Scanner(new File(console.nextLine()));
15         System.out.println();
16
17         ArrayList<String> list1 = getWords(input1);
18         ArrayList<String> list2 = getWords(input2);
19         ArrayList<String> overlap = getOverlap(list1, list2);
20
21         reportResults(list1, list2, overlap);
22     }
23
24     // post: reads all words from the given Scanner, turning them to lowercase
25     //       and returning a sorted list of the vocabulary of the file
26     public static ArrayList<String> getWords(Scanner input) {
27         // read all words and sort
28         ArrayList<String> words = new ArrayList<String>();
29         while (input.hasNext()) {
30             String next = input.next().toLowerCase();
31             words.add(next);
32         }
33         Collections.sort(words);
34
35         // add unique words to new list and return
36         ArrayList<String> result = new ArrayList<String>();
37         if (words.size() > 0) {
38             result.add(words.get(0));
39             for (int i = 1; i < words.size(); i++) {
40                 if (!words.get(i).equals(words.get(i - 1))) {
41                     result.add(words.get(i));
42                 }
43             }
44         }
45         return result;
46     }
47
48     // pre : list1 and list2 are sorted and contain no duplicates
49     // post: constructs and returns an ArrayList containing the words
50     //       in common betweeen list1 and list2
51     public static ArrayList<String> getOverlap(ArrayList<String> list1,
52                                                ArrayList<String> list2) {
53         ArrayList<String> result = new ArrayList<String>();
54         int index1 = 0;
55         int index2 = 0;
56         while (index1 < list1.size() && index2 < list2.size()) {
57             int comparison = list1.get(index1).compareTo(list2.get(index2));
58             if (comparison == 0) {
59                 result.add(list1.get(index1));
60                 index1++;
```

```
61                  index2++;
62              } else if (comparison < 0) {
63                  index1++;
64              } else {  // comparison > 0
65                  index2++;
66              }
67          }
68          return result;
69      }
70
71      // post: explains program to user
72      public static void giveIntro() {
73          System.out.println("This program compares the vocabulary of two");
74          System.out.println("text files, reporting the number of words");
75          System.out.println("in common and the percent of overlap.");
76          System.out.println();
77      }
78
79      // pre : overlap contains the words in commmon between list1 and list2
80      // post: reports statistics about two word lists and their overlap
81      public static void reportResults(ArrayList<String> list1,
82                                       ArrayList<String> list2,
83                                       ArrayList<String> overlap) {
84          System.out.println("file #1 words = " + list1.size());
85          System.out.println("file #2 words = " + list2.size());
86          System.out.println("common words  = " + overlap.size());
87
88          double percent1 = 100.0 * overlap.size() / list1.size();
89          double percent2 = 100.0 * overlap.size() / list2.size();
90          System.out.println("% of file 1 in overlap = " + percent1);
91          System.out.println("% of file 2 in overlap = " + percent2);
92      }
93  }
```

Below is an execution of the program that compares the text of Shakespeare's *Hamlet* with the text of Shakespeare's *Lear*:

```
This program compares the vocabulary of two
text files, reporting the number of words
in common and the percent of overlap.

file #1 name? hamlet.txt
file #2 name? lear.txt

file #1 words = 7234
file #2 words = 6497
common words  = 2482
% of file 1 in overlap = 34.310201824716614
% of file 2 in overlap = 38.20224719101124
```

Below is a second execution that compares the text of *Moby Dick* to the text of *Hamlet*:

```
This program compares the vocabulary of two
text files, reporting the number of words
in common and the percent of overlap.

file #1 name? moby.txt
file #2 name? hamlet.txt

file #1 words = 30368
file #2 words = 7234
common words  = 3679
% of file 1 in overlap = 12.11472602739726
% of file 2 in overlap = 50.85706386508156
```

Chapter Summary

- The ArrayList class in Java's java.util package is a class that represents a growable list of objects implemented using an array. You can use an ArrayList to store objects in a sequential order. Each element has a 0-based index.
- ArrayList is a generic class. A generic class is one that accepts a data type as a parameter when created, such as ArrayList<String>.
- An ArrayList maintains its own size for you, elements can be added and removed at any index up to the size of the list. An ArrayList can also be printed to the console. Other ArrayList operations include get, set, and clear.
- ArrayLists can be searched using methods named contains, indexOf, and lastIndexOf.
- Java's for-each loop can be used to examine each element of an ArrayList. The list can't be modified during the for-each loop.
- When storing primitive values such as int or double into an ArrayList, the list must be declared with special wrapper types such as Integer and Double.
- Objects that implement the Comparable interface can be placed into an ArrayList and sorted. Comparable defines a natural ordering for the objects of a class. The objects are compared to each other in a method named compareTo, which returns a positive, zero, or negative integer. Many common types such as String and Integer implement Comparable, and you can also implement Comparable in your own classes.

Self-Check Problems

Section 10.1: ArrayLists

1. What is an ArrayList? In what cases should you use an ArrayList rather than an array?

   The next 9 questions refer to the following String elements:

   ```
   ["It", "was", "a", "stormy", "night"]
   ```

2. Write the code to declare an ArrayList containing these elements. What is the size of the list? What is its type?

3. Write code to insert two additional elements: "dark", and "and", at the proper place in the list to produce the following ArrayList as the result:

```
["It", "was", "a", "dark", "and", "stormy", "night"]
```

4. Write code to change the second element's value to "IS", producing the following ArrayList as the result:

```
["It", "IS", "a", "dark", "and", "stormy", "night"]
```

5. Write code to remove any Strings from the list that contain the letter "a". The following should be the list's contents after your code has run:

```
["It", "IS", "stormy", "night"]
```

6. Write code to declare an ArrayList holding the first ten multiples of 2: 0, 2, 4, ..., and 18. Use a loop to fill the list with the proper elements.

7. Write a method named `maxLength` that takes an ArrayList of Strings as a parameter and that returns the length of the longest String in the list. If your method is passed an empty ArrayList, it should return 0.

8. Write code to print out whether or not a list of Strings contains the value "IS". Do not use a loop.

9. Write code to print out the index at which your list contains the value "stormy" and the index at which it contains "dark". Do not use a loop.

10. Write a for-each loop that prints the uppercase version of each String in the list on its own line.

11. When run on an ArrayList of Strings, the following code throws an exception. Why?

```
for (String s : words) {
    System.out.println(s);
    if (s.equals("hello")) {
        list.add("goodbye");
    }
}
```

12. The following code does not compile. Why not? Explain how to fix it.

```
ArrayList<int> numbers = new ArrayList<int>();
numbers.add(7);
numbers.add(19);
System.out.println(numbers);
```

13. What is a wrapper class? Describe the difference between an int and an Integer.

Section 10.2: The Comparable Interface

14. Describe how to arrange an ArrayList into sorted order. What must be true about the type of elements in the list to be able to sort it?

15. What is a natural ordering? How do you define a natural ordering for a class you've written?

16. Given the following variable declarations:

```
Integer n1 = 15;
Integer n2 = 7;
Integer n3 = 15;
String s1 = "computer";
String s2 = "soda";
String s3 = "pencil";
```

For each of the following comparisons, describe whether its result is positive, negative, or 0.

```
n1.compareTo(n2)
n3.compareTo(n1)
n2.compareTo(n1)
s1.compareTo(s2)
s3.compareTo(s1)
s2.compareTo(s2)
```

17. Write code that reads two names from the console and prints which comes first in alphabetical order using the compareTo method. For example:

```
Type a name: Tyler Durden
Type a name: Marla Singer
Marla Singer goes before Tyler Durden
```

18. Write code to read a line of input from the user and print the words of that line in sorted order. For example:

```
Type a message to sort: to be or not to be that is the question
Your message sorted: be be is not or question that the to to
```

Exercises

1. Write a method named `averageVowels` that takes an ArrayList of Strings as a parameter and that returns the average number of vowel characters (a, e, i, o, u) in all Strings in the list. If your method is passed an empty ArrayList, it should return 0.0.

2. Write a method named `swapPairs` that switches the order of values in an ArrayList of Strings in a pairwise fashion. Your method should switch the order of the first two values, then switch the order of the next two, switch the order of the next two, and so on. If there are an odd number of values in the list, the final element is not moved. For example, if the list initially stores ["to", "be", "or", "not", "to", "be", "hamlet"], your method would change the list's contents to ["be", "to", "not", "or", "be", "to", "hamlet"].

3. Write a method named `removeEvenLength` that takes an ArrayList of Strings as a parameter and that removes all of the Strings of even length from the list.

4. Write a method named `stutter` that takes an ArrayList of Strings as a parameter and that replaces every String with two of that String. For example, if the list stores the values ["how", "are", "you?"] before the method is called, it should store the values ["how", "how", "are", "are", "you?", "you?"] after the method finishes executing.

5. Write a method named `stutterK` that takes an ArrayList of integers as a parameter and that replaces every integer of value K with K copies of itself. For example, if the list stores the values [4, 1, 2, 0, 3] before the method is called, it should store the values [4, 4, 4, 4, 1, 2, 2, 3, 3, 3] after the method finishes executing. Zeroes and negative numbers should be removed from the list by this method.

6. Write a static method minToFront that takes an ArrayList of ints as a parameter and that moves the minimum value in the list to the front, otherwise preserving the order of the elements. For example, if a variable called list stores [3, 8, 92, 4, 2, 17, 9], the value 2 is the minimum, so your method would modify the list to store the values [2, 3, 8, 92, 4, 17, 9].

7. Write a method named `removeDuplicates` that takes as a parameter a sorted ArrayList of Strings and that eliminates any duplicates from the list. For example, if the list stores the values ["be", "be", "is", "not", "or", "question", "that", "the", "to", "to"] before the method is called, it should store the values ["be", "is", "not", "or", "question", "that", "the", "to"] after the method finishes executing. Because the values will be sorted, all of the duplicates will be grouped together. You may assume that the ArrayList contains only String values, but it might be empty.

8. Write a method named `removeZeroes` that takes as a parameter an ArrayList of integers and that eliminates any occurrences of the number 0 from the list. For example, if the list stores the values [0, 7, 2, 0, 0, 4, 0] before the method is called, it should store the values [7, 2, 4] after the method finishes executing.
9. Write a method named `rangeBetweenZeroes` that takes as a parameter an ArrayList of integers and returns how many indexes apart are the two furthest occurrences of the number 0. For example, if the list stores the values [7, 2, 0, 0, 4, 0, 9, 0, 6, 4, 8] when the method is called, it should return 6, because the furthest occurrences of 0 are at indexes 2 and 7, and the range 2 through 7 has 6 elements. If only one 0 occurs in the list, your method should return 1. If no 0s occur, your method should return 0.

10. Write a method named `removeInRange` that accepts three parameters: An ArrayList of Strings, a beginning String, and an ending String, and removes any Strings from the list that are alphabetically between the start and end Strings. For example, if the list contains the elements ["to", "be", "or", "not", "to", "be", "that", "is", "the", "question"] and the method is passed this list, "free" as the start String, and "rich" as the end String, the list's elements will be changed to ["to", "be", "to", "be", "that", "the"]; "or", "not", "is", and "question" would be removed because they occur alphabetically between "free" and "rich". You may assume that the start String comes earlier alphabetically than the ending String.

11. Modify the Point class from Chapter 8 so that it defines a natural ordering by implementing the Comparable interface. Compare the Points by Y-major order; that is, points with smaller Y-coordinate values come before those with higher Y-coordinate values. Break ties by comparing X-coordinate values.

12. Modify the CalendarDate class from this chapter to include a year data field, and modify its compareTo method to take years into account when making comparisons. Years take precedence over months, which take precedence over days. For example, July 18, 1995 comes before March 2, 2001.

Programming Projects

1. Write classes to model a shopping list. Make an Item class that represents a grocery item's name and price, such as tissue for $3.00. Also implement an ItemOrder class that represents a shopper's desire to purchase a given item in a given quantity, such as 5 boxes of tissue. You might wish to implement bulk-discounted items, such as 2 boxes of tissue for $4.00, which would make the given item order of (2 + 2 + 1) boxes tissue cost ($4 + $4 + $3) or $11.00. Lastly implement a ShoppingCart class that stores ItemOrders in an ArrayList and allows item orders to be added, removed, or searched for in the cart. The cart should be able to report the total price of all item orders it currently carries.

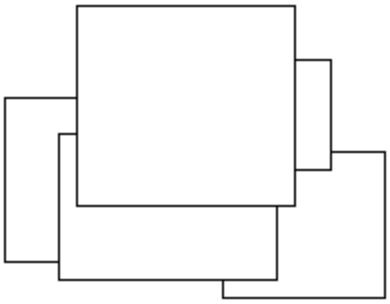
2. Write a program to reverse the lines of a file and also to reverse the order of the words in each line of the file. You should use ArrayLists to help you.

3. Write a family database program. Create a class to represent a person, who stores references to his/her mother and father, and a list of children. Read a file of names to initialize the persons' names and parent-child relationships. (You might wish to create a file representing your own family tree.) Store the overall list of persons as an ArrayList. Write an overall main user interface that asks for a name and prints a maternal and paternal family line for that person.

   Here's a hypothetical output dialogue from the program, using an input file of the English Tudor monarch line:

```
Person's name? Henry VIII
Maternal line:
    Henry VIII
        Elizabeth of York
Paternal line:
    Henry VIII
        Henry VII
Children:
    Mary I
    Elizabeth I
    Edward VI
```

4. Write a class that models a list of possibly overlapping rectangular 2D window regions, like the windows for the programs of your operating system. The order of the rectangles in the list implies the order they'd display on screen (sometimes called the "Z-order"), from 0 on the bottom to size() - 1 on the top.



Each rectangle stores its x/y position, width, and height. Your rectangle list class should have a method that takes a Point as a parameter, and treats it as though the user clicked that Point on the screen, and moves the topmost rectangle touching that point to the front of the list.

Stuart Reges
Marty Stepp

Chapter 11
Java Collections Framework

Copyright © 2006 by Stuart Reges and Marty Stepp

- 11.1 Lists
  - Collections
  - LinkedList versus ArrayList
  - Iterators
  - LinkedList Example: Sieve
  - Abstract Data Types (ADTs)
- 11.2 Sets
  - Set Concepts
  - TreeSet versus HashSet
  - Set Operations
  - Set Example: Lottery
- 11.3 Maps
  - Basic Map Operations
  - Map Views (keySet and values)
  - TreeMap versus HashMap
  - Map Example: WordCount
  - Collection Overview

Introduction

The previous chapter explored the ArrayList class. ArrayLists are one of many ways to store data in Java. In this chapter we'll explore Java's framework of different data storage collections, including linked lists, sets, and maps. We'll see how to use these structures together to manipulate and examine data in many ways to solve programming problems.

We'll see a new type of list called a linked list that stores its data differently than an ArrayList but supports the same operations. We'll also see collections called sets that don't allow duplicate elements and are good for searching. Another collection type we'll see is the map, which creates associations between pairs of data values. We'll also learn about the notion of abstract data types (ADTs) as a way to separate the capabilities of a collection from the details of its implementation.

11.1 Lists

The ArrayList class from Chapter 10 has several advantages over an array: it keeps track of its size for you, it allows insertion and removal at arbitrary places in the array, and it resizes for you if it gets full.

In this section we'll learn about an object named LinkedList which is similar to ArrayList. We'll also explore some concepts about generalizing collections and a useful object named an iterator that lets us examine the elements of any collection.

Collections

An ArrayList is an example of a *collection*, which is an object that stores a group of other objects, which are called the *elements* of the collection. A collection is sometimes also called a *data structure*.

Collection

An object that stores a group of other objects called elements.

Collections are differentiated by the types of elements they store, the operations they allow you to perform on the elements, and the speed or efficiency of those operations. Some examples of collections are:

- List: An ordered collection of elements numbered with indexes.
- Stack: A collection where elements may only be added and removed at the "top" end, allowing for Last-In, First-Out (LIFO) element access order.
- Queue: A collection where elements may only be added to the "back" end and only may be removed from the "front" end, allowing for First-In, First-Out (FIFO) element access order.
- Set: A group of elements that may not contain duplicates, and generally may be searched very quickly.
- Map: A set of pairwise mappings between a set of elements called keys and a collection of other elements named values.

Java provides a large group of useful collections that allow you to store, access, search, sort, and manipulate data in a variety of ways. Together, these collections and classes are known as the *Java Collections Framework*. This framework is largely contained in the package java.util.

There's an interface in the java.util package named Collection that every collection implements. This interface specifies the operations that most collections support. The following is a quick list of those operations:

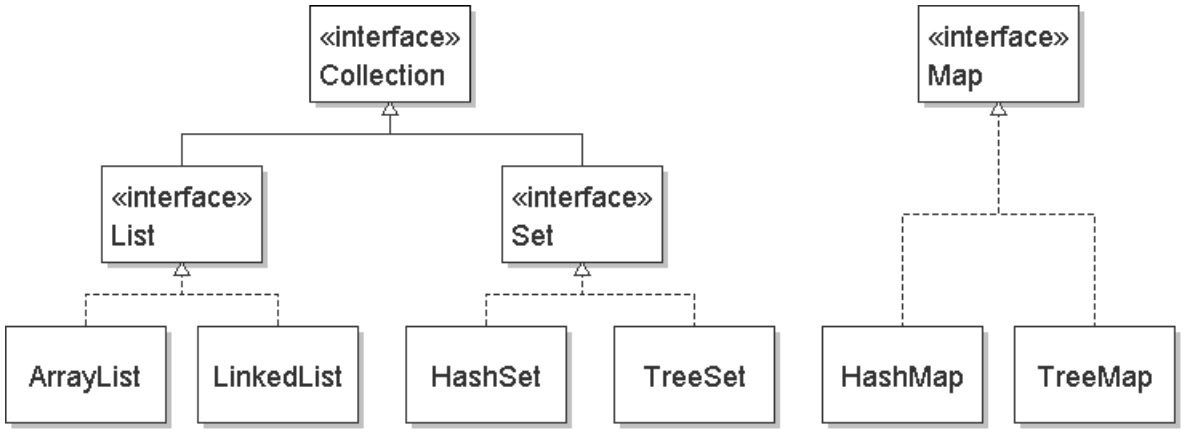
Useful Methods of the Collection Interface

Method	Description
`add(element)`	ensures that this collection contains the specified element
`addAll(collection)`	adds all elements from the given collection to this collection
`clear()`	removes all elements from this collection
`contains(element)`	returns true if this collection contains the given element

`containsAll(collection)`	returns true if this collection contains all elements of the given collection
`isEmpty()`	returns true if this collection contains no elements
`iterator()`	returns an object that helps traverse the elements of this collection
`remove(element)`	removes one occurrence of the specified element, if it is contained in this collection
`removeAll(collection)`	removes all elements of the given collection from this collection
`retainAll(collection)`	removes all elements not found in the given collection from this collection
`size()`	returns the number of elements in this collection
`toArray()`	returns an array containing the elements of this collection

The Collection interface is extended and implemented by the other interfaces and classes in the Java Collection Framework. The following diagram summarizes the various interfaces and what classes implement them. These classes will be discussed in this chapter.

An abridged view of the Java Collections Framework.



Now we'll look at a collection named LinkedList and compare it to our existing knowledge of ArrayList.

LinkedList versus ArrayList

ArrayList is a powerful and useful collection, but there are some cases where ArrayLists aren't ideal. For example, consider a program that wants to remove each String of even length from an ArrayList of strings. A straightforward way to implement such an operation might be the following:

```
// Removes all strings of even length from
// the given array list.
public static void removeEvenLength(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i += 2) {
        String element = list.get(i);
        if (element.length() % 2 == 0) {
            list.remove(i);
            i--;
        }
    }
}
```

The preceding code is correct, but it turns out that it doesn't perform well when the list has a lot of elements. It takes several minutes to process a list of a million elements. The reason it's so slow is that every time we remove an element from the list, we have to shift all subsequent elements to the left by one. Doing this many times adds up to cause a slow program.

Another case when an ArrayList behaves slowly is when we want to use it to model a waiting line or *queue*, where elements are always added at the end of the list and always removed from the front. Pseudocode for such a queue might look like this:

```
if (a customer has arrived) {
    list.add(customer).
} else {
    first customer in line = list.remove(0).
    process first customer.
}
```

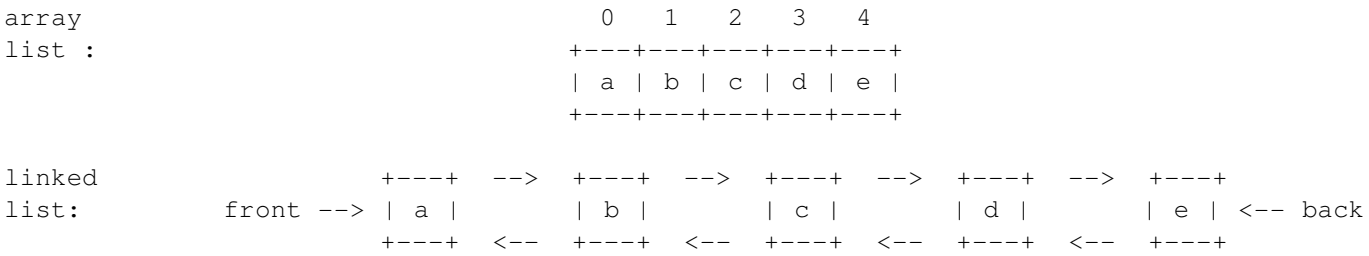
Each removal is a slow operation on a large list, because we have to slide all elements to the left by one.

There's another type of collection called a *linked list* that can give better performance in problems like these that involve a lot of adding or removal from the front or middle of a list. A linked list provides the same operations as an array list, such as add, remove, isEmpty, size, and contains. But a linked list stores its elements in a fundamentally different way. Rather than in an array, elements are stored into small individual containers named *nodes*. The nodes are "linked" together by having each node store a references to the next node in the list. The overall linked list object keeps references to the front and back nodes.

Linked List

A collection that stores a list of elements in small object containers named nodes, which are linked together by references.

One way to think of a linked list is as an array list "broken apart" where each element is stored in a small box (a node) connected to a neighboring box by an arrow.



One major advantage of a linked list is that elements can generally be added at the front or back of the list quickly, because rather than sliding many elements in an array, the list just creates a new node object and links it to the others in the list. We don't have to do this ourselves manually; the list takes care of it for us internally. The following diagram shows what happens inside the linked list when an element is added.

Adding an element to the front of a linked list.

```
   list.add(0, "x");

1. make a new node to hold the new element
          +---+
          | x |
          +---+

2. connect the new node to the other nodes in the list
          +---+
          | x |
          +---+

          |   ^
          |   |
          v   |

          +---+  -->  +---+  -->  +---+  -->  +---+  -->  +---+
front --> | a |       | b |       | c |       | d |       | e | <-- back
          +---+  <--  +---+  <--  +---+  <--  +---+  <--  +---+

3. change front of list to point to the new node

          +---+  -->  +---+  -->  +---+  -->  +---+  -->  +---+  -->  +---+
front --> | x |       | a |       | b |       | c |       | d |       | e | <-- back
          +---+  <--  +---+  <--  +---+  <--  +---+  <--  +---+  <--  +---+
```

To use a linked list in Java, create an object of type LinkedList instead of type ArrayList. LinkedList objects have the same methods you've used in ArrayList.

```
LinkedList<String> words = new LinkedList<String>();
words.add("hello");
words.add("goodbye");
words.add("this");
words.add("that");
```

Iterators

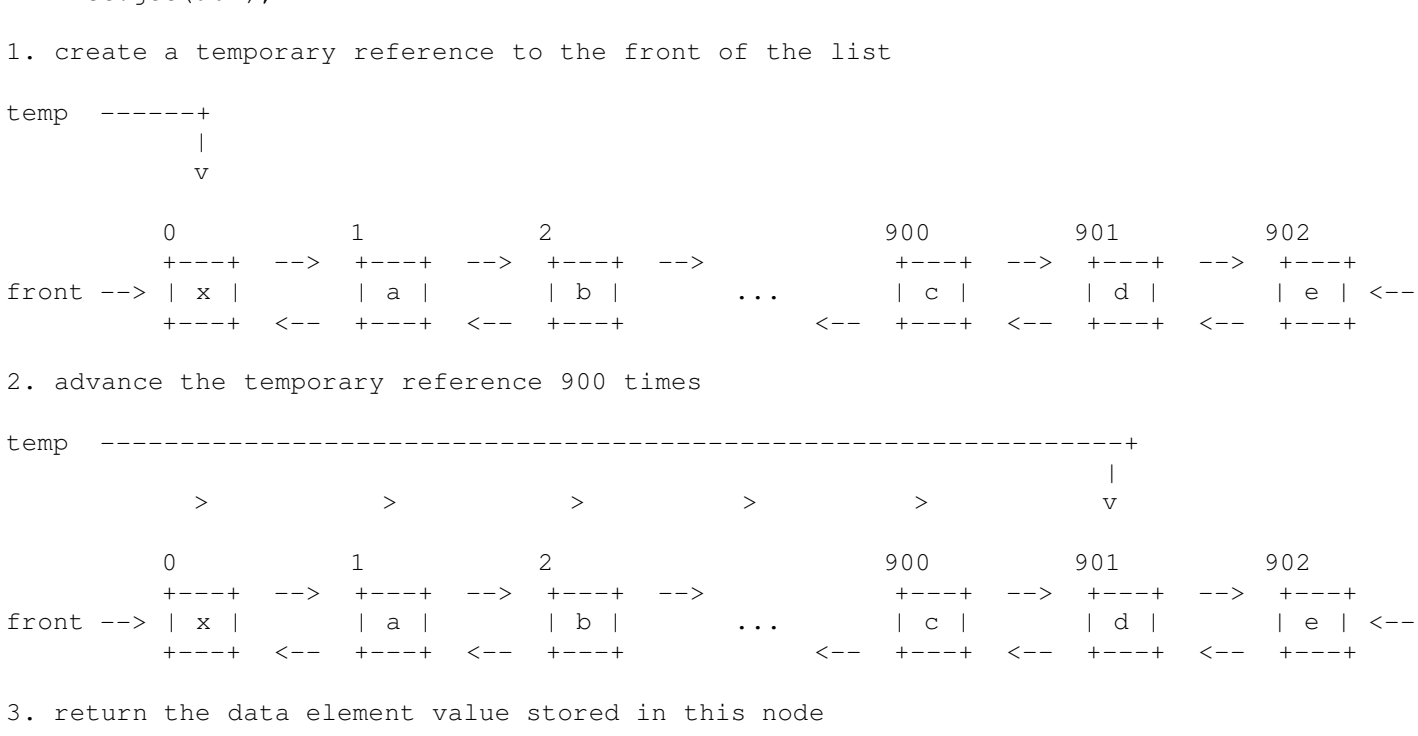
The major drawback to linked lists is that they don't provide fast *random access*; that is, you can't examine an arbitrary element of the list efficiently. The reason is because a linked list cannot jump directly to the element at a given index as you can with an array; the only way it can reach an arbitrary element is by starting at the front of the list and traversing the list nodes' "next" references to get to the index you desired.

The code that the LinkedList uses to implement its get method, for example, looks roughly like this:

```
get(int i) {
    Node temp = front.
    while (temp has not reached element i) {
        temp = temp's next.
    }
    return temp's element.
}
```

Every time you call get, set, add, or remove on a linked list, internally the list's code runs a loop that advances to that index. This means that these methods perform slowly on a LinkedList, especially if you call them many times or call them on a list with many elements. The following diagram shows the behavior of a large linked list when retrieving the element at index 901:

Retrieving an element from an arbitrary point in a linked list.



The problem is compounded when you call get (or other methods that accept an index as a parameter) multiple times, such as on each element of the linked list. If after the above call of get(901) we decided to call get(902), the linked list would again have to start from the front and advance to index 902. The only case where these methods run quickly is when you pass an index near the front or back of the list, such as 0 or the list's size minus 1.

Earlier we saw a method to remove Strings of even length from an ArrayList. If we adapted this code to use a LinkedList and made no other modifications, it would runs very slowly on large lists, because it calls the get and remove methods many times.

```
// This code performs poorly on a linked list.
public static void removeEvenLength(LinkedList<String> list) {
    for (int i = 0; i < list.size(); i++) {
        String element = list.get(i);        // slow!
        if (element.length() % 2 == 0) {
            list.remove(i);                  // slow!
            i--;
        }
    }
}
```

However, there's an efficient way to examine every element of a linked list if we want sequential access, that is, to examine each element in order from the front to the back. To do this, we can use a special object named an *iterator* that keeps track of our current position in the list, so that when we go from one element to the next, we don't have to walk the pointers all the way from the front of the list.

Iterator

An object that represents a position within a list and allows you to retrieve the elements of a list in sequential order.

As we'll see later in this chapter, iterators are very useful because every collection in the Java Collections Framework has an iterator you can use. This means that you'll have a familiar interface for examining the elements of any collection. An iterator object has the following methods:

Useful Methods of Iterator Objects

Method	Description
`hasNext()`	returns true if there are more elements to be examined
`next()`	returns the next element from the list, and advances the position of the iterator by one
`remove()`	removes the element most recently returned by next()

To get an iterator from most collections such as ArrayLists or LinkedLists, you can call the method named "iterator" on the list, which returns an Iterator object for that list's elements. (You don't use the "new" keyword.) Generally the pattern of using an iterator looks like the following:

```
Iterator<<type>> itr = <collection>.iterator();
while (itr.hasNext()) {
    do something with itr.next();
}
```

The example of removing strings of even length can be implemented much more efficiently using an iterator. While the original ArrayList version took several minutes to process a list of one million elements, this new code finishes a million-element list in under one tenth of a second. It performs this quickly because the iterator retains the current position in the list between getting or removing each element:

```
// Removes all strings of even length from
// the given linked list.
public static void removeEvenLength(LinkedList<String> list) {
    Iterator<String> i = list.iterator();
    while (i.hasNext()) {
        String element = i.next();
        if (element.length() % 2 == 0) {
            i.remove();
        }
    }
}
```

Common Programming Error: Calling next on iterator too many times

Iterators can be a bit confusing to new programmers, so you have to be careful to use them correctly. The following code attempts to use an iterator to find and return the longest String in a linked list, but it has a bug.

```
// Returns the longest string in the list.  (does not work!)
public static String longest(LinkedList<String> list) {
    Iterator<String> itr = list.iterator();
    String longest = list.next();   // initialize to first element

    while (itr.hasNext()) {
        if (itr.next().length() > longest.length()) {
            longest = itr.next();
        }
    }
    return longest;
}
```

The problem with the previous code is that the next method is called on the iterator in two places: once when testing its length, and again when trying to store the string as the longest. Each time you call next, the iterator advances by one position, so if it's called twice in the loop, you'll skip an element when you find a match. For example, if the list contains ["oh", "hello", "how", "are", "you"], we might see the "hello" and intend to store it, but the second call to next would cause us to actually store the following element "how".

The solution is to save the result of the itr.next() call into a variable. The following code would replace the previous while loop:

```
    // This version of the code is now correct.
    while (itr.hasNext()) {
        String current = itr.next();
        if (current.length() > longest.length()) {
            longest = current;
        }
    }
```

Iterators are also used internally by Java's enhanced for loop. It turns out that when you use a "for each" loop like the following, Java is actually grabbing the elements using an iterator under the hood:

```
for (String word : list) {
    System.out.println(word + " " + word.length());
}
```

As the compiler processes the for-each loop, it essentially converts it into the following code:

```
Iterator<String> i = list.iterator();
while (i.hasNext()) {
    String word = i.next();
    System.out.println(word + " " + word.length());
}
```

There's a more advanced version of Iterator named ListIterator that works only on lists. ListIterator provides operations like adding elements, setting element values, and reverse iteration from back to front. We won't discuss ListIterator in detail here.

In summary, the following are some of the major benefits of ArrayList and LinkedList:

ArrayList strengths:

- random access: any element can be accessed quickly
- adding and removing at the end of the list is fast

LinkedList strengths:

- adding and removing at either end of the list is fast
- fast add/remove during a sequential access with an iterator
- no need to expand an array when full
- can be more easily used as a 'queue'

LinkedList Example: Sieve

Consider the task of finding all prime numbers up to a given maximum. Recall that prime numbers are integers with no factors other than 1 and themselves. 2 is defined as the smallest prime number.

To build a list of prime numbers, we could certainly just write a brute-force solution using for loops:

```
for (each number from 2 to maximum) {
    if (number is prime) {
        add to list of prime numbers.
    }
}
```

To figure out whether each number is prime, we'd have to write another for loop that tested all lower integers to see whether they were factors of that number.

There's a more clever algorithm for finding prime numbers called the *Sieve of Eratosthenes*. The sieve algorithm starts by creating two lists of numbers: one list of numbers to process, and another of numbers known to be prime. Initially the numbers to process can contain every number from 2 to the maximum, while the list of primes will be empty. Here is the initial state of the two lists for a maximum of 25:

```
numbers: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
          15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
primes:  []
```

The sieve works by repeatedly removing the first element from the 'numbers' list and assuming it to be prime and then filtering out all other elements from the numbers list that are multiples of this first element. Here are the states of the lists after the first three passes of the algorithm.

```
numbers: [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25]
primes:  [2]

numbers: [5, 7, 11, 13, 17, 19, 23, 25]
primes:  [2, 3]

numbers: [7, 11, 13, 17, 19, 23]
primes:  [2, 3, 5]
```

Now let's implement the sieve algorithm. We'll use linked lists to represent the lists of numbers and primes. This is preferable over ArrayLists because we're doing a lot of removal from the front of the list. This is more efficient on a LinkedList than on an ArrayList because the ArrayList has to shift elements left on removal.

First we'll create an empty list of primes, and a list of all numbers up to the given maximum:

```
LinkedList<Integer> primes = new LinkedList<Integer>();
LinkedList<Integer> numbers = new LinkedList<Integer>();
for (int i = 2; i <= max; i++) {
    numbers.add(i);
}
```

Next we'll process the list of numbers as previously described. We'll use an iterator to make passes over the numbers list and remove elements that are multiples of the front element.

```
while (!numbers.isEmpty()) {
    // remove a prime number from the front of the list
    int front = numbers.remove(0);
    primes.add(front);

    // remove all multiples of this prime number
    Iterator<Integer> itr = numbers.iterator();
    while (itr.hasNext()) {
        int current = itr.next();
        if (current % front == 0) {
            itr.remove();
        }
    }
}
```

Here's the complete program. The main addition is a main method that prompts the user for the maximum number to examine.

```
1  // Uses a linked list to implement the Sieve of
2  // Eratosthenes algorithm for finding prime numbers.
3
4  import java.util.*;
5
6  public class Sieve {
7      public static void main(String[] args) {
8          System.out.println("This program will tell you all prime");
9          System.out.println("numbers up to a given maximum.");
10         System.out.println();
11
12         Scanner console = new Scanner(System.in);
13         System.out.print("Maximum number? ");
14         int max = console.nextInt();
15
16         LinkedList<Integer> primes = sieve(max);
17         System.out.println("Prime numbers up to " + max + ":");
18         System.out.println(primes);
19     }
20
21     // Returns a list of all prime numbers up to the given maximum
22     // using the Sieve of Eratosthenes algorithm.
23     public static LinkedList<Integer> sieve(int max) {
24         LinkedList<Integer> primes = new LinkedList<Integer>();
25
26         // add all numbers from 2 to max to a list
27         LinkedList<Integer> numbers = new LinkedList<Integer>();
28         for (int i = 2; i <= max; i++) {
29             numbers.add(i);
30         }
31
32         while (!numbers.isEmpty()) {
33             // remove a prime number from the front of the list
34             int front = numbers.remove(0);
35             primes.add(front);
36
37             // remove all multiples of this prime number
38             Iterator<Integer> itr = numbers.iterator();
39             while (itr.hasNext()) {
40                 int current = itr.next();
41                 if (current % front == 0) {
42                     itr.remove();
43                 }
44             }
45         }
46
47         return primes;
48     }
49 }
```

Here's the output from an example program that prompts the user for a maximum number and then calls the sieve method appropriately.

```
This program will tell you all prime
numbers up to a given maximum.

Maximum number? 50
Prime numbers up to 50:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

The algorithm can be optimized in a few ways. For one, the initial list of numbers doesn't actually need to store every integer from 2 through the maximum. It can instead store 2 and each odd integer up to the maximum. This makes the algorithm more efficient because it requires fewer numbers to be processed. Another optimization to the sieve algorithm is to stop once the front element of the numbers list is greater than the square root of the maximum, because any number this large that remains in the list cannot have any multiples remaining in the list. For example, in the preceding sample lists for a maximum of 25, once the front of the numbers list exceeds 5, all remaining numbers in the list are known to be prime and can be placed into the primes list. These optimizations are left as exercises.

Abstract Data Types (ADTs)

It's no accident that the LinkedList collection provides the same methods as the ArrayList. They're both considered to be implementations of the same overall type of data structure: a *list*. In fact, there's an interface in the java.util package named List that is implemented by both LinkedList and ArrayList. The List interface declares all the common methods that both lists implement. While the two list types have different benefits, the way that they're used is the same.

This idea of gathering data structures under a common interface is related to a classic data structure concept called an *abstract data type* or ADT. Formally, an ADT is a description of a data structure that specifies the type of data to be stored in the structure and the operations that can be performed on the collection. An ADT doesn't specify exactly how the data structure is implemented, only what data it can store and what it can do.

Abstract Data Type (ADT)

A specification of a data structure based on its type of data and the operations it can perform.

In Java, an ADT's set of operations is specified by the methods of the interface, and the type of data is specified by the type parameter between < and >. It's considered a good practice to declare any variables and parameters of collection types using the appropriate ADT interface type rather than the actual class type.

You can't create an object of type List, but you can use List as the declared type of a variable. For example:

```
List<Integer> list = new LinkedList<Integer>();
```

You can also use the ADT interface types like List when declaring parameters, return types, or data fields. It's useful when writing a method that accepts a collection as a parameter, because it means that method will be able to operate successfully on any collection that implements that ADT interface. For example, the following method can accept a LinkedList or ArrayList as its actual parameter:

```
// Returns the longest string in the given list.
public static String longest(List<String> list) {
```

Several useful static methods that operate on Lists are in the Collections class. These methods' headers specify parameters of type List rather than LinkedList or ArrayList. These methods perform common tasks on lists such as sorting, shuffling, and searching. Here's a quick list of useful methods from the Collections class that operate on lists:

Useful Static Methods of the Collections Class

Method	Description
`binarySearch(list, value)`	Searches a sorted array for a given element value and returns its index
`copy(destinationList, sourceList)`	Copies all elements from the source list to the destination list
`fill(list, value)`	Replaces every element in the given list with the given value
`max(list)`	Returns the element with the highest value
`min(list)`	Returns the element with the lowest value
`replaceAll(list, oldValue, newValue)`	Replaces all occurrences of the old value with the new value
`reverse(list)`	Places the elements into the opposite order
`rotate(list, distance)`	Slides each element to the right by the given number of indexes, wrapping around the last elements to the front
`shuffle(list)`	Places the elements into random order
`sort(list)`	Places the elements into sorted (nondecreasing) order
`swap(list, index1, index2)`	Switches the element values at the given two indexes

Notice that these methods are static, so they must be called by writing the word Collections followed by a dot and the method's name. For example, if we have a LinkedList variable named "list" and we wish to reverse the list's contents, we'd write:

```
Collections.reverse(list);
```

There are several other ADT interfaces in the Collections Framework besides List, such as Queue, Set, and Map. We'll visit most of them in the rest of this chapter.

11.2 Sets

A major limitation of both types of lists is that they're slow to search. Generally if we want to search a list, we have to sequentially look at each element to see if we've found the target. This can take a long time for a large list.

Another limitation of lists is that it's not easy to prevent a list from storing duplicate values. In many cases this isn't a problem, but if for example you are storing a collection to count the number of unique words in a book, you don't want any duplicates to exist. To prevent duplicates in a list, we'd have to again sequentially search the list on every add operation to make sure we weren't adding a word that was already present.

There's another abstract data type called a *set* that is very useful for maintaining a collection of elements that prevents duplicates and can be searched quickly.

> **Set**
>
> A collection that stores a group of elements without allowing duplicates.

This is very much like the mathematical notion of a set. Sets lose some abilities we had with lists, namely any operation that requires an index. But we gain new features in trade, such as elimination of duplicates and fast searching.

Set Concepts

The Collections Framework has an interface named Set whose two primary implementations are named HashSet and TreeSet. HashSet is the general-purpose set class, while TreeSet offers a few advantages seen later. If we'd like to store a set of String values, we could write the following code. The set will only contain four elements, because "Moe" would only be placed into the set once:

```
Set<String> stooges = new HashSet<String>();
stooges.add("Larry");
stooges.add("Moe");
stooges.add("Curly");
stooges.add("Moe");   // duplicate, won't be added
stooges.add("Shemp");
stooges.add("Moe");   // duplicate, won't be added
```

Sets provide exactly the operations from the Collection interface shown earlier in this chapter, such as add, contains, and remove. It's generally assumed that these operations are efficient, so that you can add many elements to a set and search it many times without incurring poor performance. A Set also provides a toString method that lets you see its elements. Printing the preceding stooges set would produce the following output:

```
[Moe, Shemp, Larry, Curly]
```

A HashSet is implemented using a special internal array named a *hash table* that places elements into specific positions based upon integers called *hash codes*. (Every Java object has a hash code that can be accessed through its hashCode method.) You don't need to understand the details of HashSet's implementation to use it, but the bottom line is that it's implemented in such a way that you can add, remove, and search for elements very quickly.

One drawback of the HashSet is that it stores its elements in an unpredictable order. The preceding stooges set's elements weren't alphabetized, nor did they match the order in which they were inserted. This strange behavior is a tradeoff for the HashSet's fast performance.

Sets are very useful for examining lots of data while ignoring duplicates. For example, if we wanted to see how many unique words appear in the book Moby Dick, we could write code such as the following:

```
Set<String> words = new HashSet<String>();
Scanner in = new Scanner(new File("mobydick.txt"));
while (in.hasNext()) {
    String word = in.next();
    word = word.toLowerCase();
    words.add(word);
}
System.out.println("Number of unique words = " + words.size());
```

The code produces the following output when run on our copy of the text of Moby Dick:

```
Number of unique words = 19474
```

Sets have a convenient constructor that accepts another collection as a parameter and puts all unique elements from that collection into the Set. One clever usage of this constructor is to find out whether a List contains any duplicates. To do so, simply construct a Set from it and see if the sizes differ. The following code demonstrates this:

```
// Returns true if the given list contains any duplicate elements.
public static boolean hasDuplicates(List<Integer> list) {
    Set<Integer> set = new HashSet<Integer>(list);
    return set.size() < list.size();
}
```

One drawback of a Set is that it doesn't store elements by indexes. So the following sort of loop doesn't compile on a Set, because it doesn't have a get method that accepts an index as a parameter:

```
// This code doesn't compile.
for (int i = 0; i < set.size(); i++) {
    String word = set.get(i);   // error -- no get method
    System.out.println(word);
}
```

Instead, if you want to loop over the elements of a Set, you must do so using an Iterator. Like all collections, sets have the iterator() method to grab an iterator for their elements. You can then use the familiar hasNext/next loop to examine each element.

```
// This code does work correctly.
Iterator<String> itr = set.iterator();
while (itr.hasNext()) {
    String word = itr.next();
    System.out.println(word);
}
```

A shorter alternative to the preceding code is to use a for-each loop over the elements of the set. As mentioned previously, the code behaves the same way but is easier to write and read.

```
for (String word : set) {
    System.out.println(word);
}
```

TreeSet versus HashSet

There's another class that implements the Set interface, named TreeSet. TreeSets store their elements using an internal linked data structure called a *binary tree*. A TreeSet is a bit slower than a HashSet, but it stores its elements in sorted order. (A TreeSet is still quite fast and is perfectly efficient enough for heavy use.)

TreeSets can be useful if you want to print the set and have the output ordered. For example, the following code displays the sorted set of all three-letter words in Moby Dick that start with "a":

```
Set<String> words = new TreeSet<String>();
Scanner in = new Scanner(new File("mobydick.txt"));
while (in.hasNext()) {
    String word = in.next();
    word = word.toLowerCase();
    if (word.startsWith("a") && word.length() == 3) {
        words.add(word);
    }
}
System.out.println("Three-letter 'a' words = " + words);
```

The code produces the following output:

```
Three-letter 'a' words = [act, add, ado, aft, age, ago, ahi,
aid, aim, air, alb, ale, all, and, ant, any, ape, apt, arc,
are, ark, arm, art, asa, ash, ask, ass, ate, awe, axe, aye]
```

A TreeSet can only be used if it knows how to sort its elements into order. This means it will work if its elements are of any type that implements the Comparable interface, such as Integer or String. You can also use TreeSets by providing your own object that specifies how to compare elements, called a Comparator. Comparators are discussed in Chapter 13 when we cover searching and sorting.

For example, you should not try to construct a TreeSet of Point elements as in the following code.

```
// Bad code!  It's illegal to use a TreeSet<Point> here,
// because Point objects are not Comparable.
Set<Point> set = new TreeSet<Point>();
set.add(new Point(5, 2));
set.add(new Point(2, 1));
```

The preceding code compiles (unfortunately), but crashes when you run it, because it doesn't know how to order the Points in the TreeSet. We'd be better off using a HashSet in this case.

```
Exception in thread "main" java.lang.ClassCastException: java.awt.Point
        at java.util.TreeMap.compare(Unknown Source)
        at java.util.TreeMap.put(Unknown Source)
        at java.util.TreeSet.add(Unknown Source)
```

In summary, the following are some of the major benefits of HashSet and TreeSet:

HashSet strengths:

- extremely fast performance for add, contains, remove
- can be used with any type of objects as its elements

TreeSet strengths:

- elements are stored in sorted order
- must be used with elements that can be compared (such as Integer, String)

Set Operations

Consider the case where you have two sets and need to figure out how many total unique elements occur between the two sets. You can't just add the sets' sizes, since they might have some elements in common that shouldn't be counted twice in your total. You could count all elements from the first set and count only the unique elements of the second, by checking to see whether each element from the second is also in the first:

```
// Returns the number of elements contained in both set1 and set2.
public static int totalElements(Set<String> set1, Set<String> set2) {
    int count = set1.size();
    for (String element : set2) {
        if (!set2.contains(element)) {
            count++;
        }
    }
    return count;
}
```

A more elegant way to perform this calculation is to compute a *union* between the the sets. The union of two sets A and B is the set of all elements that are in contained in either A or B or both. Union is an example of a *set operation*; a set operation is a combination of two sets to produce a new set or result. Other set operations include *intersection* (the set of all elements that are in both A and B) and *difference* (the set of all elements that are in A but not in B).

You can write code to perform set operations by calling the various "All" methods with the relevant pair of sets. The following table summarizes which method corresponds to which set operation:

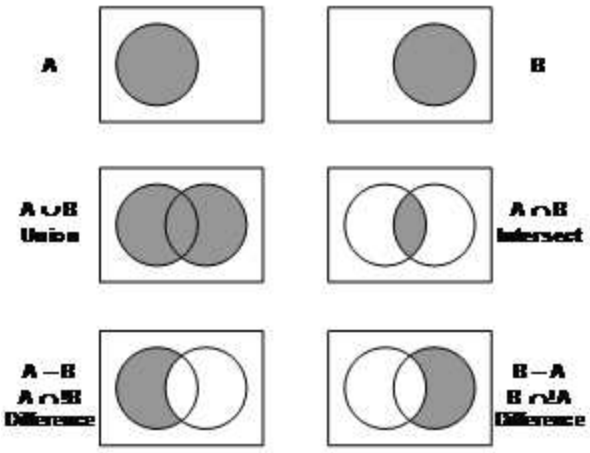
Set operation	Method	Description
union	`addAll`	set of all elements that are in A or B or both
intersection	`retainAll`	set of all elements that are in both A and B
difference	`removeAll`	set of all elements that are in A but not in B
superset	`containsAll`	returns true if A is a superset of (contains all elements of) B

For example, one could rewrite the totalElements code to use a union with the addAll method:

```
// Returns the number of elements contained in both set1 and set2.
public static int totalElements(Set<String> set1, Set<String> set2) {
    Set<String> union = new HashSet<String>(set1);
    union.addAll(set2);
    return union.size();
}
```

An important thing to note about the set operations in Java is that they modify the existing set rather than creating a new set for you. Notice that in the preceding totalElements code, we initialize a new HashSet that contains all elements from set1 and then add set2's contents to our new set, rather than combining set1 and set2 directly. We do this because the caller might not want us to disturb their sets' original contents.

Set operations are often depicted by drawings called Venn diagrams, where sets are circles and shaded overlapping between circles represents set operations. Here's an example:



Set Example: Lottery

Consider the task of writing a lottery program. The program should randomly generate a winning lottery ticket, then prompt the player to enter their lotto numbers. Depending on how many numbers match, the player wins various cash prizes.

Sets make excellent collections for storing the winning lotto numbers and the player's numbers. They prevent the possibility of duplicates, and they quickly allow us to test whether a number in one set is in the other. This will help us to count the number of correct winning numbers the player has entered.

The following code uses a Random object to initialize a set of 6 winning lottery numbers between 1 and 40. The code uses a while loop because the same number might be randomly generated more than once.

```
Set<Integer> winningNumbers = new TreeSet<Integer>();
Random r = new Random();
while (winningNumbers.size() < 6) {
    int number = r.nextInt(40) + 1;
    winningNumbers.add(number);
}
```

Once the winning number set is generated, we'll read the player's lotto numbers into a second set. To figure out how many numbers the player has chosen correctly, we could search the winning number set to see whether it contains each number from the ticket. But a more elegant way to perform this test is to perform an intersection between the winning numbers set and the player's ticket set. The following code creates the intersection of the player's ticket and the winning numbers, by copying the ticket and then removing any elements from it that aren't winning numbers:

```
// find the winning numbers from the user's ticket
Set<Integer> intersection = new TreeSet<Integer>(ticket);
intersection.retainAll(winningNumbers);
```

Once we have the intersection, we can ask for its size to see how many of the player's numbers were winning numbers, and we can pick an appropriate cash prize amount for them. (Our version starts with a $100 prize and doubles for each winning number.)

With all of this in mind, here is the complete code for the lottery program. We've made a few static methods for structure and made a few constants for the number of numbers, maximum number, and lotto prize amounts.

```
1  // Plays a quick lottery game with the user,
2  // reading lucky numbers and printing how many
3  // matched a winning lottery ticket.
4
5  import java.util.*;
6
7  public class Lottery {
8      public static final int NUMBERS = 6;
9      public static final int MAX_NUMBER = 40;
10
11     public static void main(String[] args) {
12         // get winning number and ticket sets
13         Set<Integer> winningNumbers = createWinningNumbers();
14         Set<Integer> ticket = getTicket();
15         System.out.println();
16
17         // keep only the winning numbers from the user's ticket
18         Set<Integer> intersection = new TreeSet<Integer>(ticket);
19         intersection.retainAll(winningNumbers);
20
21         // print results
22         System.out.println("Your ticket numbers are " + ticket);
23         System.out.println("The winning numbers are " + winningNumbers);
24         System.out.println();
25         System.out.println("You had " + intersection.size() + " matching numbers.");
26         if (intersection.size() > 0) {
27             double prize = 100 * Math.pow(2, intersection.size());
28             System.out.println("The matched numbers are " + intersection);
29             System.out.println("Your prize is $" + prize);
30         }
31     }
32
33     // generates a set of the winning lotto numbers
34     public static Set<Integer> createWinningNumbers() {
35         Set<Integer> winningNumbers = new TreeSet<Integer>();
36         Random r = new Random();
37         while (winningNumbers.size() < NUMBERS) {
38             int number = r.nextInt(MAX_NUMBER) + 1;
39             winningNumbers.add(number);
40         }
41         return winningNumbers;
42     }
43
44     // reads the player's lottery ticket from the console
45     public static Set<Integer> getTicket() {
46         Set<Integer> ticket = new TreeSet<Integer>();
47         Scanner console = new Scanner(System.in);
48         System.out.print("Type your " + NUMBERS + " unique lotto numbers: ");
49         while (ticket.size() < NUMBERS) {
50             int number = console.nextInt();
51             ticket.add(number);
52         }
53         return ticket;
54     }
55 }
```

Here's one example output from running the program:

```
Type your 6 unique lotto numbers: 2 8 15 18 21 32

Your ticket numbers are [2, 8, 15, 18, 21, 32]
The winning numbers are [1, 3, 15, 16, 18, 39]

You had 2 matching numbers.
The matched numbers are [15, 18]
Your prize is $400.0
```

11.3 Maps

Consider the task of writing a telephone book program where users can type a person's name and search for that person's phone number. You could store the data into an array, list, or set. Perhaps you'd make a small class named PhoneBookRecord that stores a person's name and phone number; your list would contain many PhoneBookRecord objects. When searching for a phone number, you'd traverse the list and look for the PhoneBookRecord with that name, and return its phone number.

A solution such as the one just described isn't very practical in practice. In a large list of records, it takes the program a long time to look at each one to find the right record to retrieve the phone number.

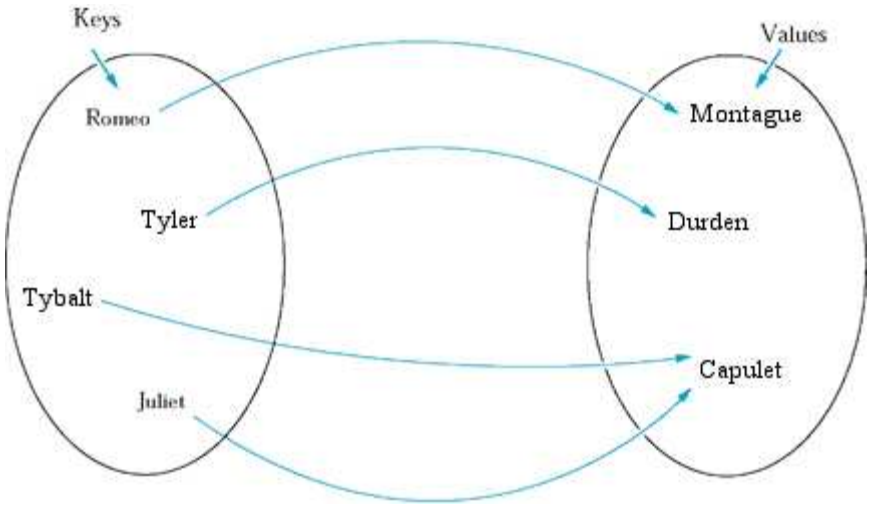
There are many data processing tasks like this in which it's useful to link pairs of objects (like a name and a telephone number) together. We find ourselves saying, "I'd like to associate every A with a B." Here are some examples of tasks like this:

- Store a collection of names and phone numbers. When the user types a name, look up that person's phone number.
- Count the number of occurrences of every word in a book.
- Group the words of a book by length and report how many words are each length.
- Associate each chat user with their set of friends / buddies.
- Store a family tree where each person knows about his/her mother and father.

Java's collection framework includes an abstract data type called map that helps to solve these types of tasks. A *map* allows you to create many one-way associations between pairs of objects. We say that a map associates a *key* with a *value*. A key can only map to one value, but it's possible for multiple keys to map to the same value.

> **Map**
>
> A data structure that associates objects called keys with objects called values.

A map can be thought of as a pair of connected collections: a set of keys and a collection of values associated with those keys. The following diagram is an example of this idea:



Basic Map Operations

A Map is constructed with not one but two generic type parameters, separated by commas. The first type parameter represents the type of the keys, and the second represents the type of the values. This makes for a lengthy declaration line in your code. The following is an example of constructing a salary map (which associates people's names with their salaries). Notice that we have to use the wrapper type Double rather than the primitive type double.

```
Map<String, Double> salaryMap = new HashMap<String, Double>();
```

There is an interface named Map to represent the Map ADT; the two primary classes that implement the interface are named HashMap and TreeMap. HashMap is the more general-purpose map, and TreeMap stores comparable keys in sorted order.

One main operation on a map is the put method, which is roughly similar to the add method of most other collections. The put method accepts a key and a value as parameters, and stores a mapping between the key and value into the map. (If the key was previously associated with some other value, it is replaced.)

```
salaryMap.put("Stuart Reges", 20000.00);
salaryMap.put("Marty Stepp", 15500.00);
salaryMap.put("Jenny", 86753.09);
```

Once you've added a key/value pair to the map, you can look up a value later by calling the map's get method, which accepts a key as a parameter and returns the value associated with that key.

```
double jenSalary = salaryMap.get("Jenny");
System.out.println("Jenny's salary is $" + jenSalary);
```

To see whether a map contains a mapping for a given key, you can use the containsKey method, or you can call the get method and test for a null result.

```
Scanner console = new Scanner(System.in);
System.out.print("Type a person's name: ");
String name = console.nextLine();

// search the map for the given name
if (salaryMap.containsKey(name)) {
    double salary = salaryMap.get(name);
    System.out.println(name + "'s salary is $" + salary + ".");
} else {
    System.out.println("I don't have a salary record for " + name + ".");
}
```

Some people think of maps as a generalization of lists. In a list, you store elements associated with integer indexes. In a map, you store elements associated with object keys. A map is somewhat like a list except that you can use objects as the "indexes" for storing and retrieving elements.

The following is a list of the methods of Maps.

Useful Methods of Maps

Method	Description
`clear()`	
`containsKey(key)`	returns true if the given key maps to some value in this map
`containsValue(value)`	returns true if some key maps to the given value in this map
`get(key)`	returns the value associated with this key, or null if not found
`isEmpty()`	returns true if this collection contains no keys or values
`keySet()`	returns a Set of all keys in this map
`put(key, value)`	associates the given key with the given value
`putAll(map)`	adds all key/value mappings from the given map to this map
`remove(key)`	removes the given key and its associated value from this map
`size()`	returns the number of key/value mappings in this map
`values()`	returns a Collection of all values in this map

Maps do have a toString method that represents them as each key followed by an = sign, followed by the value to which that key maps. The order in which the keys appear depends on the type of map used, which we'll get to in a moment. Here's what the salary map declared previously would look like when printed:

```
{Jenny=86753.09, Stuart Reges=20000.0, Marty Stepp=15500.0}
```

Map Views (keySet and values)

Unlike most collections, a map doesn't have an iterator method, because it wouldn't be clear exactly what you wanted to iterate over. The keys? The values? Both? Instead, maps have a pair of methods named keySet and values that respectively return you a Set of all keys in the map and a Collection of all values in the map. These are sometimes called *collection views* of a map because they are each collections that conceptually exist inside of the map..

For example, consider a map whose keys are persons' names and whose values are their Social Security Numbers.

```
Map<String, Integer> ssnMap = new HashMap<String, Integer>();
ssnMap.put("Stuart Reges", 439876305);
ssnMap.put("Marty Stepp", 504386382);
ssnMap.put("Jenny", 867530912);
```

If we wanted to write a loop that printed every person's name who is in the map, we could call the keySet method on the map. This method returns a Set collection containing every key from the hash table -- in this case, every String for a person's name. If you store the keySet into a variable, you should declare that variable as type Set, with the map's keys' type between the < and > .

```
Set<String> nameSet = ssnMap.keySet();
for (String name : nameSet) {
    System.out.println("Name: " + name);
}
```

The preceding code would produce the following output (the keys are in an unpredictable order since a HashMap is used):

```
Name: Jenny
Name: Stuart Reges
Name: Marty Stepp
```

If we instead wanted to loop over every Social Security Number (every value) stored in the map, we'd instead call the values method on the map. The values method returns a reference of type Collection and not of type Set, because the values may contain duplicates since it's legal for two keys to map to the same value. If you store the keySet into a variable, you should declare that variable as type Collection, with the map's values' type between the < and > .

```
Collection<Integer> ssnValues = ssnMap.values();
for (int ssn : ssnValues) {
    System.out.println("SSN: " + ssn);
}
```

The preceding code would produce the following output:

```
SSN: 867530912
SSN: 439876305
SSN: 504386382
```

The keys and values are often combined by looping over the keys and then getting the value for each key. Notice that the following code also doesn't declare a variable to store the key set, but instead just calls keySet directly in the for-each loop.

```
for (String name : ssnMap.keySet()) {
    int ssn = ssnMap.get(name);
    System.out.println(name + "'s SSN is " + ssn);
}
```

The resulting output is the following:

```
Jenny's SSN is 867530912
Stuart Reges's SSN is 439876305
Marty Stepp's SSN is 504386382
```

There's a related method named entrySet that returns objects of a type called Entry that represents both the keys and the values, but we won't explore this here.

TreeMap versus HashMap

Similarly to Sets which we saw previously, there are two flavors of Map collections in Java: HashMap and TreeMap. As with sets, a HashSet performs a bit faster and can store any type of data, and it keeps its keys in a somewhat haphazard order. By contrast, a TreeMap can only store Comparable data and is a bit slower, but it keeps its keys in sorted order.

If we declared the Social Security Number map from the previous section to use a TreeMap, we'd see a different ordering to the keys when we printed it on the console.

```
Map<String, Integer> ssnMap = new TreeMap<String, Integer>();
ssnMap.put("Stuart Reges", 439876305);
ssnMap.put("Marty Stepp", 504386382);
ssnMap.put("Jenny", 867530912);
System.out.println(map);
```

The following output would be produced. Notice that the names (the map's keys) are in sorted alphabetical order.

```
{Jenny=867530912, Marty Stepp=504386382, Stuart Reges=439876305}
```

HashMap is still the one recommended for general use by Sun, because many applications don't care about the order of the keys, because many programs benefit from its better performance, and because it works even on data that isn't Comparable and therefore doesn't have any natural ordering.

Map Example: Word Count

In an earlier example we counted the number of unique words in the book Moby Dick. What if we were asked to find the words in the book that occur the most times? It seems like for each word in the book, that word has a count of how many times it occurs. If we could examine all of those counts and print the ones with large values, we'd have our answer.

It turns out that maps are very useful for solving such a problem. We can maintain a word-count map where each key is a word and its associated value is the number of occurrences of that word in the book.

```
wordCountMap = empty.
for (each word from file) {
    if (I have never seen this word before) {
        set this word's count to 1.
    } else {
        increase this word's count by one.
    }
}
```

Assuming that we have a Scanner to read the appropriate file and a Map to store word counts, like the following:

```
Map<String, Integer> wordCountMap = new TreeMap<String, Integer>();
Scanner in = new Scanner(new File("mobydick.txt"));
```

We could read the file's contents and store them in the map as follows. Notice that if the word has been seen before, we retrieve its old count value, increment it by 1, then put the new value back into the map. When you put a key/value mapping into a map that already contains that key, the old mapping is replaced. For example, if the word "ocean" mapped to the number 25 and we put a new mapping from "ocean" to 26, the old mapping from "ocean" to 25 is replaced. We don't have to remove it manually.

```
while (in.hasNext()) {
    String word = in.next();
    if (!wordCountMap.containsKey(word)) {   // never seen before
        map.put(word, 1);
    } else {                                 // seen before
        int count = map.get(word);
        map.put(word, count + 1);
    }
}
```

Once we've built the word count map, if we wanted to print all words that appear more than, say, 2000 times in the book, we could write code like the following:

```
for (String word : wordCountMap.keySet()) {
    int count = wordCountMap.get(word);
    if (count > 2000) {
        System.out.println(word + " occurs " + count + " times.");
    }
}
```

Here's the complete program, with a method added for structure and a constant for the number of occurrences needed for a word to be printed.

```
1  // Uses maps to implement a word count, so that the user
2  // can see which words occur the most in the book Moby Dick.
3
4  import java.io.*;
5  import java.util.*;
6
7  public class WordCount {
8      // minimum number of occurrences needed to be printed
9      public static final int OCCURRENCES = 2000;
10
11     public static void main(String[] args) throws FileNotFoundException {
12         System.out.println("This program displays the most frequently");
13         System.out.println("occurring words from the book Moby Dick.");
14         System.out.println();
15
16         // read the book into a map
17         Scanner in = new Scanner(new File("mobydick.txt"));
18         Map<String, Integer> wordCountMap = getCountMap(in);
19
20         for (String word : wordCountMap.keySet()) {
21             int count = wordCountMap.get(word);
22             if (count > OCCURRENCES) {
23                 System.out.println(word + " occurs " + count + " times.");
24             }
25         }
26     }
27
28     // Reads the book text and returns a map from words to counts
29     public static Map<String, Integer> getCountMap(Scanner in) {
30         Map<String, Integer> wordCountMap = new TreeMap<String, Integer>();
31
32         while (in.hasNext()) {
33             String word = in.next().toLowerCase();
34             if (!wordCountMap.containsKey(word)) {
35                 // never seen this word before
36                 wordCountMap.put(word, 1);
37             } else {
38                 // seen this word before; increment count
39                 int count = wordCountMap.get(word);
40                 wordCountMap.put(word, count + 1);
41             }
42         }
43
44         return wordCountMap;
45     }
46 }
```

The program produces the following output on our copy of Moby Dick:

```
This program displays the most frequently
occurring words from the book Moby Dick.

a occurs 4509 times.
and occurs 6138 times.
his occurs 2451 times.
in occurs 3975 times.
of occurs 6405 times.
that occurs 2705 times.
the occurs 13991 times.
to occurs 4433 times.
```

Collection Overview

We've seen three major abstract data types in this chapter: lists, sets, and maps. It's important to understand the differences between them and when each should be used. The following table summarizes the different collections and their pros and cons:

ADT	implementations	description	benefits	weaknesses	example usages
List	ArrayList, LinkedList	an ordered sequence of elements	maintains an indexed position for every element	slow to search, slow to add/ remove arbitrary elements	list of accounts; waiting line; the lines of a file
Set	HashSet, TreeSet	a set of unique elements	fast to search; no duplicates	missing indexes; cannot retrieve arbitrary elements from it	unique words in a book; lottery ticket numbers
Map	HashMap, TreeMap	a group of associations between pairs of "key" objects and "value" objects	allows you to connect one piece of information to another and search for one based on the other	harder to understand and use	word counting; phone book

When approaching a new programming problem involving data, you can ask yourself questions to help decide what collection is most appropriate. Here are some examples:

- How do you plan to search the data? If you want to look up particular elements of data by index, you want a list. If you intend to search for elements, a set may be better. If you need to find an object given partial information about it (for example, to find a user's bank account object based on its PIN or ID number), a map may be best.

- What are you going to do with the data? Do you intend to add and remove many elements, or search the data many times, or connect this data to other data?

- In what order should the elements be stored? Lists order their elements by order of insertion, while the Tree collections (TreeSet and TreeMap) order elements by their Comparable natural ordering. If order doesn't matter, you may want a Hash collection such as HashSet.

Chapter Summary

- A collection is an object that stores a group of other objects. Examples of collections are ArrayLists, HashSets, and TreeMaps. Collections are used to structure, organize, and search data.
- A linked list is a collection similar to an ArrayList but implemented internally by storing each element into a small container object named a node. Linked lists are faster than array lists at certain operations such as adding and removing from the front or middle of the list.
- An iterator is an object that represents a position in a list and lets you examine its elements in order. Linked lists are often used with iterators because it's slow to access an arbitrary element from the list.
- An abstract data type (ADT) is a specification of a data structure by what data it can hold and what operations it can perform. Two examples of ADTs are List and Set. ADTs in Java's collection framework are represented as interfaces such as the List interface, which is implemented by both LinkedList and ArrayList.
- A set is a collection that doesn't allow duplicates. Sets also generally can be searched very quickly to see whether they contain a particular element value. The Set interface represents sets.
- There are two major set classes in Java: TreeSet and HashSet. A TreeSet holds Comparable data in a sorted order; a HashSet can hold any data and can be searched more quickly, but its elements are stored in unpredictable order.
- A map is a collection that associates key objects with value objects. Maps are used to create associational relationships between pieces of data, such as associating a person's name with their phone number.
- There are two major map classes in Java: TreeSet and HashSet. A TreeMap holds Comparable keys in a sorted order; a HashMap can hold any data as its keys and performs value lookups more quickly, but its keys are stored in unpredictable order.

Self-Check Problems

Section 11.1: Lists

1. When should you use a LinkedList instead of an ArrayList?
2. Would a LinkedList or an ArrayList perform better when run on the following code? Why?

```
// This code performs very poorly on a linked list!
public static int min(LinkedList<Integer> list) {
    int min = list.get(0);
    for (int i = 1; i < list.size(); i++) {
        if (list.get(i) < min) {
            min = list.get(i);
        }
    }
    return min;
}
```

3. What is an iterator? Why are iterators often used with linked lists?

4. Write a piece of code that counts the number of duplicate elements in a linked list; that is, the number of elements whose values are repeated at an earlier index in the list. Assume that all duplicates in the list occur consecutively. For example, the list [1, 1, 3, 5, 5, 5, 5, 7, 7, 11] contains 5 duplicates: one duplicate of element value 1, three duplicates of element value 5, and one duplicate of element value 7. You may wish to put your code into a method named countDuplicates that accepts the linked list as a parameter and returns the number of duplicates.

5. Write a piece of code that inserts a String into an ordered linked list of Strings, maintaining sorted order. For the list ["Alpha", "Baker", "Foxtrot", "Tango", "Whiskey"], inserting "Charlie" in order would produce the list ["Alpha", "Baker", "Charlie", "Foxtrot", "Tango", "Whiskey"].

6. Write a method named removeAll that accepts a linked list of integers as a parameter and removes all occurrences of a particular value. You must preserve the original relative order of the remaining elements of the list. For the list [3, 9, 4, 2, 3, 8, 17, 4, 3, 18, 2, 3], the call `removeAll(list, 3)` would change the list to [9, 4, 2, 8, 17, 4, 18, 2].

7. Write a method named wrapHalf that accepts a linked list of integers as a parameter and moves the first half of the list onto the back of the list. If the list contains an odd number of elements, the smaller half is wrapped (in other words, for a list of size N, the middle element N / 2 becomes the first element in all cases). For the list [1, 2, 3, 4, 5, 6], calling wrapHalf on the list would change that list into [4, 5, 6, 1, 2, 3]. For the list [5, 6, 7, 8, 9], the result would be [7, 8, 9, 5, 6].

8. What is an abstract data type (ADT)? What ADT does a linked list implement?

9. A previous question asked you to count the duplicates in a linked list. Rewrite this as a method named countDuplicates that will allow either an ArrayList or a LinkedList to be passed as the parameter.

Section 11.2: Sets

10. Lists have every method that a Set has, and more. So why would you use a Set rather than a List?

11. When should you use a TreeSet, and when should you use a HashSet?

12. A Set doesn't have the get and set methods that an ArrayList has. So how do you examine every element of a Set?

13. What elements are contained in the following set after this code executes?

```
Set<Integer> set = new HashSet<Integer>();
set.add(74);
set.add(12);
set.add(182);
set.add(90);
set.add(43);
set.remove(74);
set.remove(999);
set.remove(43);
set.add(32);
set.add(182);
set.add(9);
set.add(29);
```

14. What elements are contained in the following set after this code executes?

```
Set<Integer> set = new HashSet<Integer>();
set.add(8);
set.add(41);
set.add(18);
set.add(50);
set.add(132);
set.add(28);
set.add(79);
set.remove(41);
set.remove(28);
set.add(86);
set.add(98);
set.remove(18);
```

15. What elements are contained in the following set after this code executes?

```
Set<Integer> set = new HashSet<Integer>();
set.add(4);
set.add(15);
set.add(73);
set.add(84);
set.add(247);
set.remove(15);
set.add(42);
set.add(12);
set.remove(73);
set.add(94);
set.add(11);
```

16. How do you perform a union operation on two sets? An intersection? Try to give an answer that doesn't require any loops.

Section 11.3: Maps

17. Write the code to declare a Map from people's names to their ages. Add mappings for your own name/age, as well as those of a few friends or relatives.

18. A Map doesn't have the get and set methods that an ArrayList has. It doesn't even have an iterator method like a Set does, nor can a for-each loop be used on it directly. So how do you examine every key (or every value) of a Map?

19. What keys and values are contained in the following map after this code executes?

```
HashMap<Object, String> map = new HashMap<Object, String>();
map.put(7, "Marty");
map.put(34, "Louann");
map.put(27, "Donald");
map.put(15, "Moshe");
map.put(84, "Larry");
map.remove("Louann");
map.put(7, "Ed");
map.put(2350, "Orlando");
map.remove(8);
map.put(5, "Moshe");
map.remove(84);
map.put(17, "Steve");
```

20. What keys and values are contained in the following map after this code executes?

```
HashMap<Integer, String> map = new HashMap<Integer, String>();
map.put(8, "Eight");
map.put(41, "Forty-one");
map.put(8, "Ocho");
map.put(18, "Eighteen");
map.put(50, "Fifty");
map.put(132, "OneThreeTwo");
map.put(28, "Twenty-eight");
map.put(79, "Seventy-nine");
map.remove(41);
map.remove(28);
map.remove("Eight");
map.put(86, "Eighty-six");
map.put(98, "Ninety-eight");
map.remove(18);
```

21. Modify the word count program so that it prints the words sorted by the number of occurrences. To do this, write code at the end of the program to create a reverse map from counts to words, based on the original map. Assume that no two words of interest occur the exact same number of times.

Exercises

1. Modify the Sieve program example from this chapter to make 2 optimizations. First, instead of storing all integers up to the maximum in the numbers list, only store 2 and all odds starting from 3 upward. Second, if the front of the numbers list ever reaches the square root of the maximum, you may put all remaining values from the numbers list into the primes list. (Why is this true?)

2. Write a method named alternate that accepts two Lists as its parameters and returns a new List containing alternating elements from the two lists in the following order:
   - first element from first list
   - first element from second list
   - second element from first list
   - second element from second list
   - third element from first list
   - third element from second list

   ...

   If the lists do not contain the same number of elements, the remaining elements from the longer list sit consecutively at the end. For a first list of [1, 2, 3, 4, 5] and a second list of [6, 7, 8, 9, 10, 11, 12], the call of `alternate(list1, list2)` should return a list containing [1, 6, 2, 7, 3, 8, 4, 9, 5, 10, 11, 12].

3. Write a method named removeInRange that accepts four parameters: a LinkedList, an element value, a starting index, and an ending index. Its behavior is to remove all occurrences of the given element that appear in the list between the starting index (inclusive) and the ending index (exclusive). Other values and occurrences of the given value that appear outside the given index range are not affected.

   For example, with the list [0, 0, 2, 0, 4, 0, 6, 0, 8, 0, 10, 0, 12, 0, 14, 0, 16], the call of `removeInRange(list, 0, 5, 13);` would produce the list [0, 0, 2, 0, 4, 6, 8, 10, 12, 0, 14, 0, 16]. Notice that the zeroes whose indexes were between 5 inclusive and 13 exclusive in the original list (before any modifications are made) are removed.

4. Write a method named partition that accepts a list of integers and an integer value E as its parameter, and rearranges (partitions) the list so that all its elements less than E occur before all elements greater than E. The exact order of the elements is unimportant so long as all elements less than E appear before all elements greater than E. For example, for the linked list [15, 1, 6, 12, -3, 4, 8, 21, 2, 30, -1, 9], one acceptable ordering of the list after the call of partition(list, 5) would be [-1, 1, 2, 4, -3, 12, 8, 21, 6, 30, 15, 9]. You may assume that the list contains no duplicates and does not contain the element value E.

5. Write a method named sortAndRemoveDuplicates that accepts a list of integers as its parameter and rearranges the list's elements into sorted ascending order, as well as removing all duplicate values from the list. For example, the list [7, 4, -9, 4, 15, 8, 27, 7, 11, -5, 32, -9, -9] would become [-9, -5, 4, 7, 8, 11, 15, 27, 32] after a call to your method. Use a Set as part of your solution.

6. Write a method named symmetricSetDifference that accepts two Sets as parameters and returns a new Set containing their symmetric set difference: the set of elements contained in either of two sets but not in both. For example, the symmetric difference between the sets {1, 4, 7, 9} and {2, 4, 5, 6, 7} is {1, 2, 5, 6, 9}.

7. Write a method named is1to1 that accepts a Map<String, String> as its parameter and returns true if no two keys map to the same value. For example, {Marty=206-9024, Hawking=123-4567, Smith=949-0504, Newton=123-4567} returns false, but {Marty=206-9024, Hawking=555-1234, Smith=949-0504, Newton=123-4567} returns true. The empty map is considered 1-to-1 and returns true.

8. Write a method named subMap that accepts two Map<String, String> as its parameters and returns true if every key in the first map is also contained in the second map, and every key in the first map maps to the same value in the second map. For example, {Smith=949-0504, Marty=206-9024} is a subMap of {Marty=206-9024, Hawking=123-4567, Smith=949-0504, Newton=123-4567}. The empty map is a subMap of every map.

9. Write a method named reverse that accepts a Map<String, String> as a parameter and returns a new Map that is the original's "reverse". The reverse of a map is a new map that uses the values from the original as its keys and the keys from the original as its values. Since a map's values need not be unique but its keys must be, you should have each value map to a set of keys. In other words, if the original map maps keys of type K to values of type V, the new map maps keys of type V to values that are Sets containing elements of type K. For example, the map {42=Marty, 81=Sue, 17=Ed, 31=Dave, 56=Ed, 3=Marty, 29=Ed} has a reverse of {Marty=[42, 3], Sue=[81], Ed=[17, 56, 29], Dave=[31]}. (The order of the keys and values does not matter.)

Programming Projects

1. Write a program that computes the edit distance between two words. The edit distance (also called Levenshtein distance, for creator Vladimir Levenshtein) between two strings is the minimum number of operations needed to transform one string into the other. For this program, an operation is a substitution of a single character, such as from "brisk" to "brick". The edit distance between the words "dog" and "cat" is 3, because the chain of "dot", "cot", and "cat" transforms "dog" into "cat". When computing the edit distance between two words, each intermediate word must be an actual valid word. Edit distances are useful in applications that need to determine how similar two strings are, such as spell checkers.

   Read your input from a dictionary text file. From this file, compute a map from every word to its immediate neighbors, that is, the words that have an edit distance of 1 from it. Once this map is built, you can walk it to find paths from one word to another.

   A good way to process paths to walk the neighbor map is to use a linked list of words to visit, starting initially with the beginning word, such as "dog". Your algorithm should repeatedly remove the front word of the list and add all of its neighbors to the end of the list, until the ending word (such as "cat") is found or until the list becomes empty, which would indicate that no path exists between the two words.

2. Write a program that solves a classic "stable marriage" problem. This problem deals wih a group of men and a group of women. The program tries to pair them up so as to generate as many marriages as possible that are all stable. A set of marriages is unstable if you can find a man and a woman who would rather be married to each other than to their current spouses (in which case, the two would be inclined to divorce their spouses and marry each other).

   The input file for the program will list all of the men, one per line, followed by a blank line, followed by all of the women, one per line. The men and women are numbered by their position in the input file (the first man is #1, the second man is #2, and so on; the first woman is #1, the second woman is #2, and so on). Each input line (except the blank line separating men from women) has a name followed by a colon followed by a list of integers. These integers are the preferences for this particular person. For example, the following input line in the men's section,

   ```
   Joe: 10 8 35 9 20 22 33 6 29 7 32 16 18 25
   ```

   indicates that the person is named "Joe" and that his first choice for marriage is woman #10, his second choice is woman #8, and so on. Any women not listed are considered unacceptable to Joe.

   The stable marriage problem is solved by the following algorithm:

   ```
   assign each person to be free.
   while (some man M with a nonempty preference list is free) {
       W = first woman on M's list.

       if (some man P is engaged to W) {
           assign P to be free.

       assign M and W to be engaged to each other.

       for (each successor Q of M who is on W's list) {
           delete W from Q's preference list.
           delete Q from W's preference list.
       }
   }
   ```

   For the following input:

   ```
   Man 1: 4 1 2 3
   Man 2: 2 3 1 4
   Man 3: 2 4 3 1
   Man 4: 3 1 4 2
   Woman 1: 4 1 3 2
   Woman 2: 1 3 2 4
   Woman 3: 1 2 3 4
   Woman 4: 4 1 3 2
   ```

   The following is a stable marriage solution:

```
Man 1 and Woman 4
Man 3 and Woman 2
Man 2 and Woman 3
Man 4 and Woman 1
```

Stuart Reges
Marty Stepp

Chapter 12
Recursion

Copyright © 2006 by Stuart Reges and Marty Stepp

- 12.1 Thinking Recursively
  - A Nonprogramming Example
  - An Iterative Solution Converted to Recursion
  - Structure of Recursive Solutions
- 12.2 A Better Example of Recursion
  - Mechanics of Recursion
- 12.3 Recursive Functions
  - Integer Exponentiation
  - Greatest Common Divisor
- 12.4 Recursive Graphics (optional)
- 12.5 Case Study: Prefix Evaluator
  - Infix, Prefix and Postfix Notation
  - Prefix Evaluator
  - Complete Program

Introduction

This chapter focuses on a programming technique known as *recursion*. It allows us to solve certain complex problems in a highly elegant manner. The chapter begins by comparing recursion to what we already know. Then it discusses the low-level mechanics that make recursion work in Java. Finally, we examine a number of problems that are easily expressed using the technique.

12.1 Thinking Recursively

The problem solving techniques we have seen so far are part of classical *iteration*, also known as the *iterative approach*.

> **Iteration (Iterative)**
>
> A programming technique in which you describe actions to be repeated using a loop.

In this chapter we will explore a new technique known as *recursion*:

> **Recursion (Recursive)**
>
> A programming technique in which you describe actions to be repeated using a method that calls itself.

You have spent so much time writing solutions iteratively that it will take a while to get used to thinking about problems recursively. Let's begin with a nonprogramming example.

A Nonprogramming Example

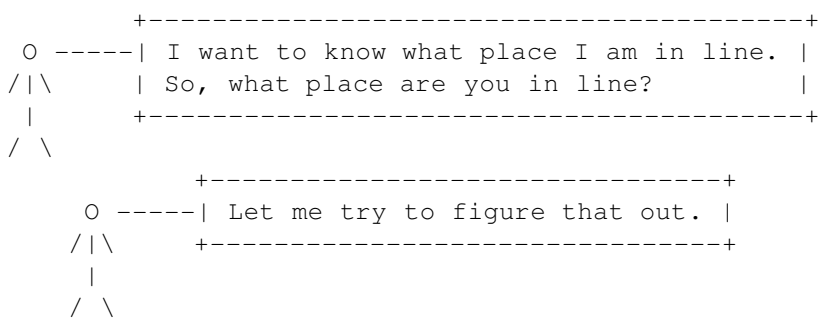
We often find ourselves standing in line. You might find yourself wondering what place you are in line. Are you number 10 in line? Are you number 20? How would you find out?

Most people would solve this problem iteratively. You'd count the people in line: one, two, three, and so on. This is like a while loop that continues while there are more people left to count. The iterative approach is a natural one, but it has some limitations. For example, what if the person in front of you is tall and you're not very tall. Will you be able to see past the person to see exactly who is in line? Or maybe the line goes around the block and you can't see around the corner to count the people there.

Can you think of another way to solve the problem? To think about the problem recursively, you have to think in terms of having all the people standing in line working together to solve the problem. So instead of having one person doing all of the counting, you have each person do a little piece of the counting.

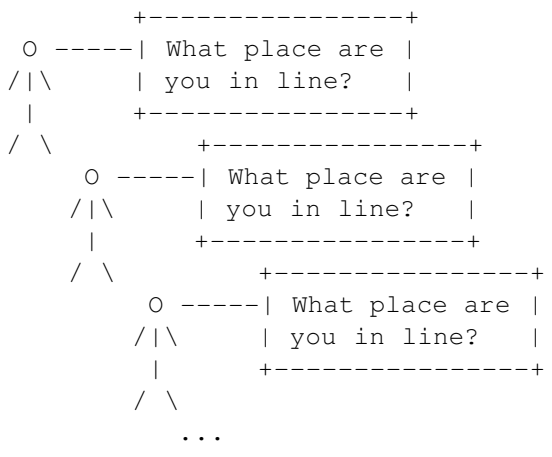
One approach would be to ask the person in front of me what my place in line is. That person might ask another person who might ask another person. But that doesn't help much because it's the exact same question. That just leads to a bunch of people saying, "This guy wants to know what place he is in line. Does anyone know?" Probably then someone would start counting and solve the problem iteratively.

We have to somehow make the problem simpler. So instead of asking the person in front of me what place *I* am in line, I should ask the person what place *he* or *she* is in line:

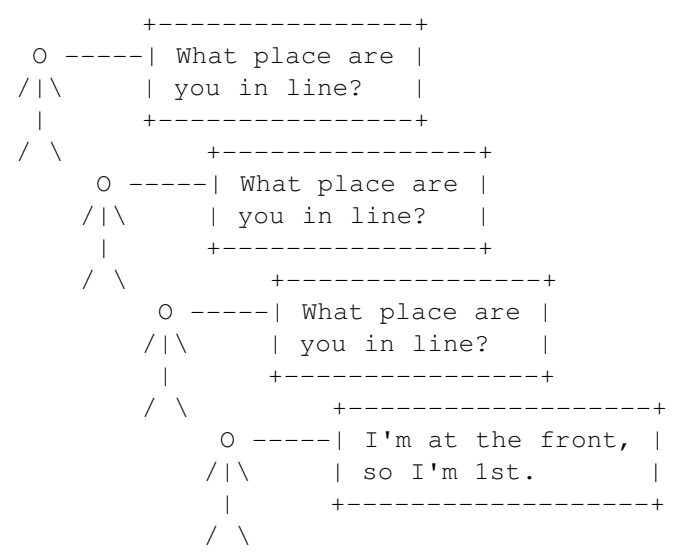


The key difference is that the person in front of me is closer to the front of the line. Suppose, for example, that I'm 10th in line. Then the person in front of me is 9th in line. That's closer to the front. But notice that I'm asking the person in front of me to think about the exact same *kind* of question I'm thinking about. We're both trying to figure out our place in line although the person in front of me is closer to the front than I am. That's where the idea of recursion comes in. The problem *recurs* in that each of us wants to answer the same question.

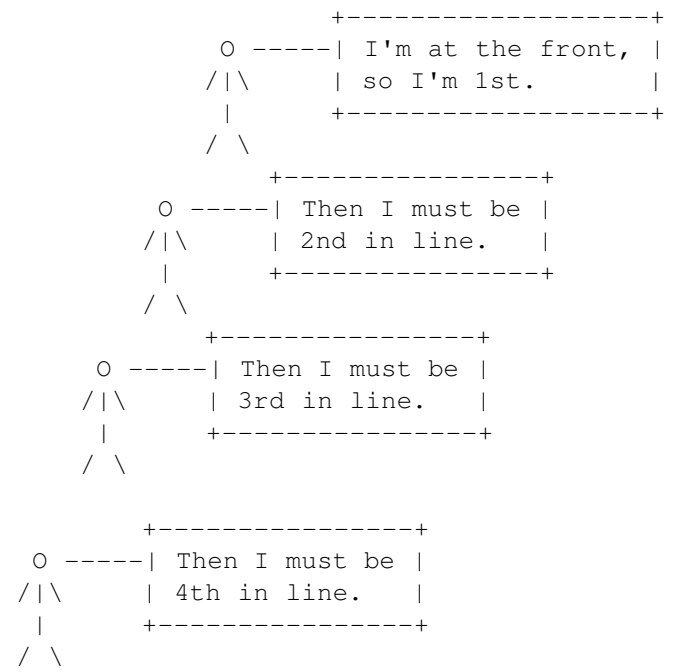
The idea is to set up a chain reaction of people all asking the person in front of them the same question:



The line has to end somewhere, so eventually this process has to end. Eventually we have to reach a person who is at the front of the line:



At this point we have reached what we sometimes refer to as the *bottom* of the recursion. We have gotten a bunch of people involved in collectively solving the problem and we've finally reached a point where we can start assembling the answer. The person at the front is in position 1. That means the person just before is at position 2 and the person just before is at position 3 and so on. So after we reach the bottom of the recursion, we *unwind* the recursion and figure out the overall answer:



These diagrams included just four individuals to make them fit on a page, but you can imagine this process working even if there were 30 people in line or 300 people in line.

One of the key aspects to notice here is that recursion involves many cooperating entities that each solve a little bit of the problem. Instead of one person who does all of the counting, we have each individual asking one question as we go towards the front of the line and answering one question as we come back out.

In programming the iterative solution of having one person doing all the counting is like having a loop that repeats some action. The recursive solution of having many people each doing a little bit of work will turn into many different method calls each of which perform a little bit of work. So let's see an example of how a simple iterative solution could be turned into a recursive solution.

An Iterative Solution Converted to Recursion

Suppose that you want to have a method called writeStars that will take an integer parameter n and that will produce a line of output with exactly n stars on it. We can solve this with a simple for loop:

```
public static void writeStars(int n) {
    for (int i = 1; i <= n; i++) {
        System.out.print("*");
    }
    System.out.println();
}
```

The action being repeated here is the call on System.out.print that prints the n stars. If we want to write this recursively, we need to think about different cases. We might ask the method to produce a line with 10 stars or we might ask the method to produce a line with 20 stars or we might ask the method to produce a line of 50 stars. Of all of the possible star-writing tasks we might ask it to perform, which one is the simplest?

Students often answer that printing a line of 1 star is very easy and they're right that it's easy. But there is a task that is even easier. Printing a line of 0 stars requires almost no work at all. We can create such a line by calling System.out.println. So we begin our recursive definition with a test for this case:

```
public static void writeStars2(int n) {
    if (n == 0) {
        System.out.println();
    } else {
        ...
    }
```

In the else part we want to deal with lines that have more than 0 stars on them. Your instinct is probably to fill in the else part with the for loop we used before. You have to fight the instinct to solve the entire problem that way. To solve this second part of the problem, it is important to think about how we can do just a small amount of work that will get us closer to being done. If the number of stars is greater than 0, then we know we have to print at least one, so we can add that to our code:

```
public static void writeStars2(int n) {
    if (n == 0) {
        System.out.println();
    } else {
        System.out.print("*");
        // what is left to do?
        ...
    }
```

At this point in the process you have to be able to make a certain leap of faith. You have to believe that recursion actually works. Once we have written a single star, what do we have left to do? The answer is that we want to write (n - 1) more stars along with a println. In other words, after writing one star, what we have left to do is to write a line of (n - 1) stars. You should find yourself thinking, "If only I had a method that would produce a line of (n - 1) stars, then I could call that method." But you *do* have such a method. It's the method we are writing. So after writing a single star, we can call the writeStars2 method itself to complete the line of output:

```
public static void writeStars2(int n) {
    if (n == 0) {
        System.out.println();
    } else {
        System.out.print("*");
        writeStars2(n - 1);
    }
}
```

Many novices complain that this seems like cheating. We are supposed to be writing the method called writeStars2, so how can we call writeStars2 from inside of writeStars2? Welcome to the world of recursion.

In our nonprogramming example we talked about having many people standing in line who solved a problem together. To understand a recursive method like writeStars2, it is useful to imagine each method invocation as being like a person. The key insight is that there isn't just one person who can do the writeStars2 task. We have an entire army of people who can each do the writeStars2 task.

Let's think about what happens when I call the method and request a line of 3 stars:

```
writeStars2(3);
```

We call up the first person from the writeStars2 army and say, "I want a line of 3 stars." The person looks at the code in the method and sees that the way you write a line of 3 stars is to:

```
System.out.print("*");
writeStars2(2);
```

In other words, the first member of the army writes a star and calls up the next member of the army to write a line of 2 stars. Just as we had a series of people figuring out what place they were in line, we have a series of people called up who each print one star and then call on someone else to write the rest of the line. With the people standing in line, we eventually reached the person at the front of the line. In this case we reach a point where the request is to write a line of 0 stars, which leads us into the if branch rather than the else branch and we complete the task with a simple println.

Below is a trace of the calls that would be made:

```
writeStars2(3); // n > 0, execute else
    System.out.print("*");
    writeStars2(2); // n > 0, execute else
        System.out.print("*");
        writeStars2(1); // n > 0, execute else
            System.out.print("*");
            writeStars2(0); // n == 0, execute if
                System.out.println();
```

So a total of four different calls are made on the method. If we think of the calls as being like people, a total of four members of the army are called up to solve the task together. Each one was solving a star-writing task, but they were solving slightly different tasks (3 stars, 2 stars, 1 star, 0 stars). This is similar to the nonprogramming example in which the various people standing in line were all answering the same kind of question but they were solving slightly different problems because of where they were standing in line (closer to the front or further from the front).

Structure of Recursive Solutions

Writing recursive solutions requires a certain leap of faith, but there is nothing magical about recursion. For example, the following method is not a solution to the task of writing a line of n stars:

```
public static void writeStars3(int n) {
    writeStars3(n);
}
```

This version produces infinite recursion. For example, if I ask the method to write a line of 10 stars, it tries to accomplish that by asking the method to write a line of 10 stars, which asks the method to write a line of 10 stars, which asks the method to write a line of 10 stars, and so on. In other words, this solution is the recursive equivalent of an infinite loop.

Every recursive solution that you write will have two key ingredients: a *base case* and a *recursive case*.

Base Case

A case that is so simple that it can be solved directly without a recursive call.

Recursive Case

A case that involves reducing the overall problem to a simpler problem of the same kind that can be solved by a recursive call.

Below is the writeStars2 method with its base case and recursive case indicated with comments:

```
public static void writeStars2(int n) {
    if (n == 0) {
        // base case
        System.out.println();
    } else {
        // recursive case
        System.out.print("*");
        writeStars2(n - 1);
    }
}
```

The base case is the task of writing a line of zero stars. This task is so simple that it can be done immediately. The recursive case is the task of writing lines with 1 or more stars. To solve the recursive case, we begin by writing a single star, which reduces the remaining task to that of writing a line of (n - 1) stars. This is a task that the writeStars2 method is designed to solve and it is simpler than the original task, so we can solve it by making a recursive call.

As an analogy, suppose that you are at the top of a ladder with n rungs on it. If you have a way to get from one rung to the one below and if you can recognize when you've reached the ground, then you can handle any height ladder. Going from one rung to the one below is like the recursive case where you perform some small amount of work that reduces the problem to a simpler one of the same form (get down from rung (n - 1) versus get down from rung n). Recognizing when you reach the ground is like the base case that can be solved directly (step off the ladder).

Some problems involve multiple base cases and some problems involve multiple recursive cases, but there will always be at least one of each. If you are missing either, you run into trouble. Without the ability to step down from one rung to the one below, you'd be stuck at the top of the ladder. Without the ability to recognize when you reach the ground, you'd keep climbing even when there are no rungs left in the ladder.

Because recursive solutions include some combination of base cases and recursive cases, you will find that they are often written with an if/else statement, a nested if statement or some minor variation. You will also find that recursive programming generally involves a case analysis where you categorize the possible forms the problem might take into different cases and you write a solution for each case.

12.2 A Better Example of Recursion

While it might be interesting to solve the writeStars task with recursion, it isn't a very compelling example. Let's look in detail at a problem where recursion simplifies the work we have to do.

Suppose that you have a Scanner that is tied to an external input file and you want to print the lines of the file in reverse order. For example, the file might contain the following four lines of text:

```
this
is
fun
no?
```

We want to print these lines in reverse order, so we'd like to produce this output:

```
no?
fun
is
this
```

To solve the problem iteratively, we'd need some kind of data structure for storing the lines of text. We might, for example, use an ArrayList<String>. With recursion we can solve the problem without using a data structure.

Remember that recursive programming involves thinking about cases. What would be the simplest file to reverse? A one-line file would be fairly easy to reverse, but it would be even easier to reverse an empty file. So we can begin writing our method as follows:

```
public static void reverse(Scanner input) {
    if (!input.hasNextLine()) {
        // base case (empty file)
        ...
    } else {
        // recursive case (nonempty file)
        ...
    }
}
```

For this problem the base case is so simple that there isn't anything to do. An empty file has no lines to reverse. So for this problem, it makes more sense to turn around the if/else so that we test for the recursive case. That way we can make it a simple if that has an implied "else there is nothing to do":

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        ...
    }
}
```

Again the challenge is to do a little bit. How do we get a bit closer to being done? We can read one line of text from the file:

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        String line = input.nextLine();
        ...
    }
}
```

So in our example, this would read the line "this" into the variable line and would leave us with these three lines of text in the Scanner:

```
is
fun
no?
```

We're trying to produce this overall output:

```
no?
fun
is
this
```

You might be asking yourself questions like, "Is there another line of input to process?" That's not how to think recursively. If you're thinking recursively, you'd think about what a call on the method would get you. Since the Scanner is positioned in front of the three lines: is/fun/no?, a call on reverse should read in those lines and produce the first three lines of output that we're looking for. If that works, then we'd just have to write out the line "this" afterwords to complete the output.

This is where the leap of faith comes in. We have to believe that the reverse method actually works. Because if it does, then this code can be completed as follows:

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        String line = input.nextLine();
        reverse(input);
        System.out.println(line);
    }
}
```

It turns out that this works. To reverse a sequence of lines you read the first one in, reverse the others, and then write the first one out. It doesn't seem that it should be that simple, but it is.

Mechanics of Recursion

Novices seem to understand recursion better when they know more about the underlying mechanics that make it work. Before we examine a recursive method in detail we should make sure we understand how nonrecursive methods work. Consider the following simple program:

```
 1  public class DrawTriangles {
 2
 3      public static void drawTriangle() {
 4          System.out.println("  *");
 5          System.out.println(" ***");
 6          System.out.println("*****");
 7          System.out.println();
 8      }
 9
10      public static void drawTwoTriangles() {
11          drawTriangle();
12          drawTriangle();
13      }
14  }
```

It simply prints three triangles:

```
  *
 ***
*****

  *
 ***
*****

  *
 ***
*****
```

How do we describe the method calls that take place in this program? Imagine that each method has been written on a different piece of paper. We begin program execution by going to method main, so imagine grabbing the sheet of paper with main on it:

```
+-------------------------------------------+
| public static void main(String[] args) {  |
|     drawTriangle();                       |
|     drawTwoTriangles();                   |
| }                                         |
+-------------------------------------------+
```

To execute the method, we go through each of the statements from first to last and execute each one. So we first hit the call on drawTriangle:

```
+-------------------------------------------+
| public static void main(String[] args) {  |
| --> drawTriangle();                       |
|     drawTwoTriangles();                   |
| }                                         |
+-------------------------------------------+
```

We know that at this point in time, the computer stops executing main and turns its attention to the drawTriangle method. A good way to think about this is that you grab the piece of paper with drawTriangle written on it and you put it over the piece of paper with main on it:

```
+-------------------------------------------+
|  public static void main(String[] args) {  |
|  +----------------------------------------+
|  | public static void drawTriangle() {    |
|  |     System.out.println("  *");         |
+--|     System.out.println(" ***");         |
   |     System.out.println("*****");        |
   |     System.out.println();               |
   | }                                       |
   +----------------------------------------+
```

Now we execute each of the statements in drawTriangle from first to last and go back to main. When we go back to main, we will have finished the call on drawTriangle, which means that the next thing to do is the call on drawTwoTriangles:

```
+-------------------------------------------+
| public static void main(String[] args) {  |
|     drawTriangle();                       |
| --> drawTwoTriangles();                   |
| }                                         |
+-------------------------------------------+
```

So now we grab the piece of paper with drawTwoTriangles on it and we place it over the paper with main on it:

```
+-------------------------------------------+
|  public static void main(String[] args) {  |
|  +----------------------------------------+
|  | public static void drawTwoTriangles() {  |
|  |     drawTriangle();                    |
+--|     drawTriangle();                    |
   | }                                      |
   +----------------------------------------+
```

The first thing to do here is the first call on drawTriangle:

```
+-----------------------------------------------+
|  public static void main(String[] args) {     |
|  +--------------------------------------------+
|  | public static void drawTwoTriangles() {    |
|  | --> drawTriangle();                        |
+--|     drawTriangle();                        |
   | }                                          |
   +--------------------------------------------+
```

To execute this method, we take out the sheet of paper with drawTriangle on it and place it on top:

```
+-----------------------------------------------+
|  public static void main(String[] args) {     |
|  +--------------------------------------------+
|  | public static void drawTwoTriangles() {    |
|  |  +-----------------------------------------+
|  |  | public static void drawTriangle() {     |
|  |  |     System.out.println("  *");          |
+--|  |     System.out.println(" ***");          |
   |  |     System.out.println("*****");         |
   +--|     System.out.println();                |
      | }                                       |
      +-----------------------------------------+
```

By putting these papers one on top of the other we can visually see that we started with method main which called method drawTwoTriangles which called method drawTriangle. So at this moment in time, three different methods are active. The one on top is the one that we are actively executing. Once we finish with it, we go back to the one underneath. And once we finish it, we'll go back to main. We could continue this example, but you probably get the idea by now.

The idea of representing each method call as a piece of paper and putting them on top of each other as they are called is a metaphor for Java's *call stack*.

> **Call Stack**
>
> The internal structure that keeps track of the sequence of methods that have been called.

If you think of the call stack as being like a stack of papers with the most recently called method on top, you have a pretty good idea of how it works.

Let's use the idea of the call stack to understand how the recursive file reversing method works. To be able to visualize the call stack, we need to put the method definition on a piece of paper:

```
+-----------------------------------------------+
|  public static void reverse(Scanner input) {  |
|      if (input.hasNextLine()) {               |
|          String line = input.nextLine();      |
|          reverse(input);                      |
|          System.out.println(line);            |
|      }                                        |
|  }                                            |
|      +------------------+                     |
| line |                  |                     |
|      +------------------+                     |
+-----------------------------------------------+
```

Notice that the paper includes a place to store the value of the local variable called "line". This detail will turn out to be very important.

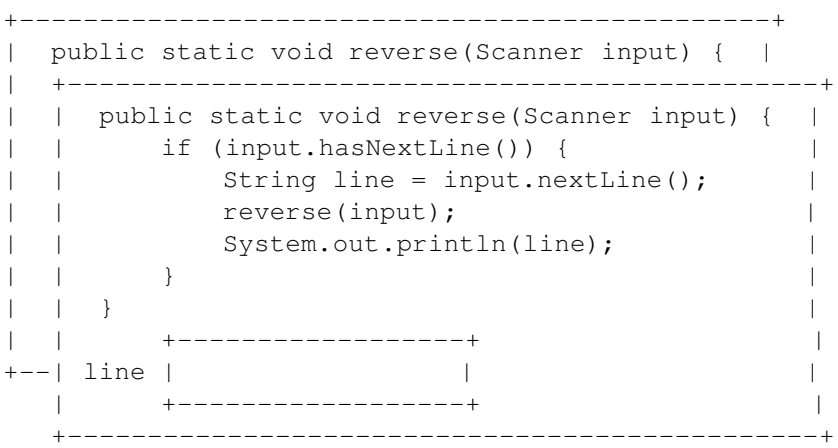
Suppose that we call this method with our sample input file with the following four lines of text:

```
this
is
fun
no?
```

When we call the method, it reads the first line of text into its variable line and then it reaches the recursive call on reverse:

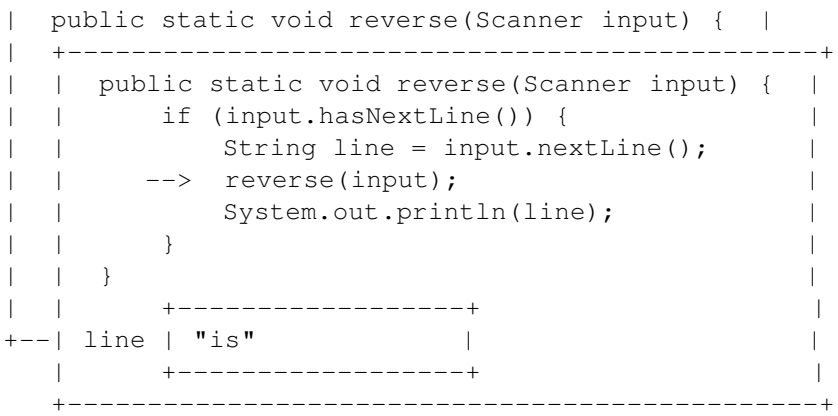
```
+-----------------------------------------------+
|  public static void reverse(Scanner input) {  |
|      if (input.hasNextLine()) {               |
|          String line = input.nextLine();      |
|     -->  reverse(input);                      |
|          System.out.println(line);            |
|      }                                        |
|  }                                            |
|      +------------------+                     |
| line | "this"           |                     |
|      +------------------+                     |
+-----------------------------------------------+
```

Then what happens? In the triangle program we grabbed the sheet of paper for the method being called and placed it on top of the current sheet of paper. Here we have method reverse calling method reverse. To understand what happens, you have to realize that each method invocation is independent of the others. We don't have just a single sheet of paper with the reverse method written on it, we have as many copies as we want. So we can grab a second copy of the method definition and place it on top of this one:

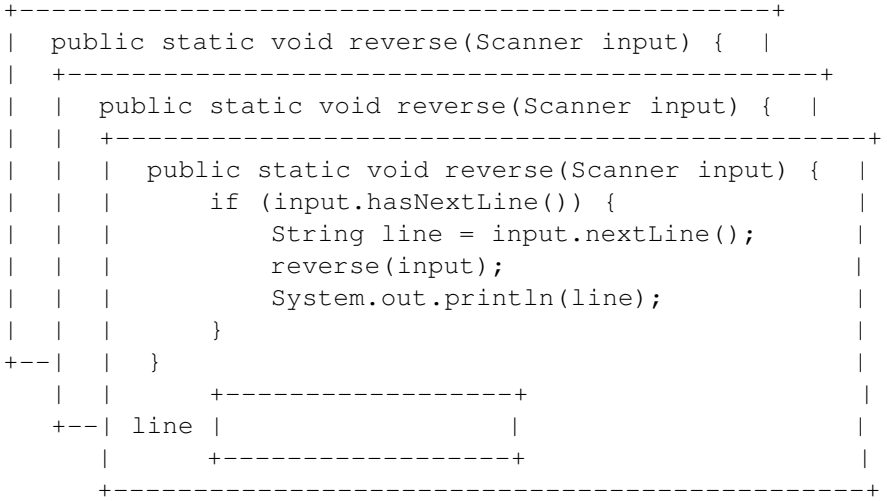


This new version of the method has its own variable called "line" in which it can store a line of text. And even though the previous version (the one underneath this one) is in the middle of its execution, this new one is at the beginning of its execution. Think back to the analogy of an entire army of robots all programmed with the same code. In the same way, we can bring up as many copies of the reverse method as we need to solve this problem.

For this second call on reverse we see that there is a line of text to be read and it reads it in (the second line that contains "is"). Then it finds itself executing a recursive call on reverse:



So Java sets aside this version of the method as well and brings up a third version:



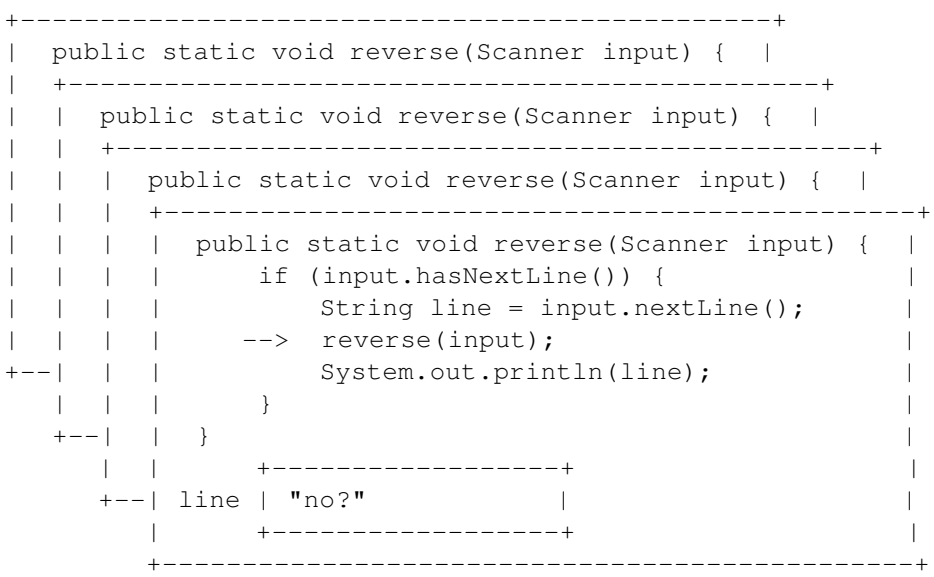
Again, notice that it has its own variable called "line" that is independent of the other variables called "line." This version of the method finds that there is a line to reverse (the third line, "fun"), so it reads it in and reaches a recursive call on reverse:

```
+-----------------------------------------------+
|  public static void reverse(Scanner input) {  |
|  +--------------------------------------------+
|  |  public static void reverse(Scanner input) {  |
|  |  +-----------------------------------------+
|  |  |  public static void reverse(Scanner input) {  |
|  |  |      if (input.hasNextLine()) {               |
|  |  |          String line = input.nextLine();      |
|  |  |     -->  reverse(input);                      |
|  |  |          System.out.println(line);            |
|  |  |      }                                        |
+--|  |  }                                            |
   |  |      +------------------+                     |
   +--| line | "fun"            |                     |
      |      +------------------+                     |
      +-----------------------------------------------+
```

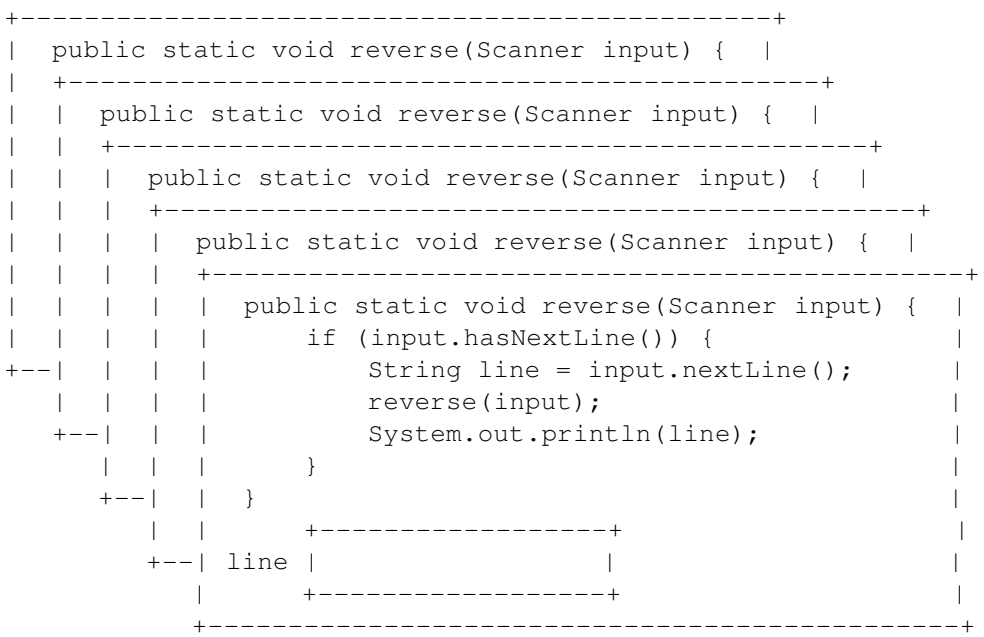
This brings up a fourth version of the method:

```
+-----------------------------------------------+
|  public static void reverse(Scanner input) {  |
|  +--------------------------------------------+
|  |  public static void reverse(Scanner input) {  |
|  |  +-----------------------------------------+
|  |  |  public static void reverse(Scanner input) {  |
|  |  |  +--------------------------------------+
|  |  |  |  public static void reverse(Scanner input) {  |
|  |  |  |      if (input.hasNextLine()) {               |
|  |  |  |          String line = input.nextLine();      |
|  |  |  |          reverse(input);                      |
+--|  |  |          System.out.println(line);            |
   |  |  |      }                                        |
   +--|  |  }                                            |
      |  |      +------------------+                     |
      +--| line |                  |                     |
         |      +------------------+                     |
         +-----------------------------------------------+
```

This one finds a fourth line of input ("no?"), so it reads that in and reaches the recursive call:

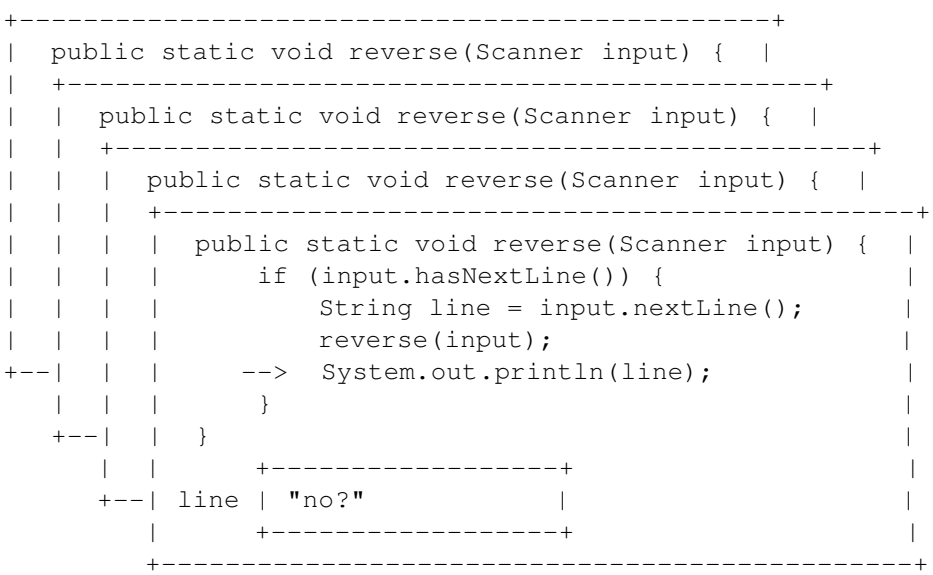


This brings up a fifth version of the method:

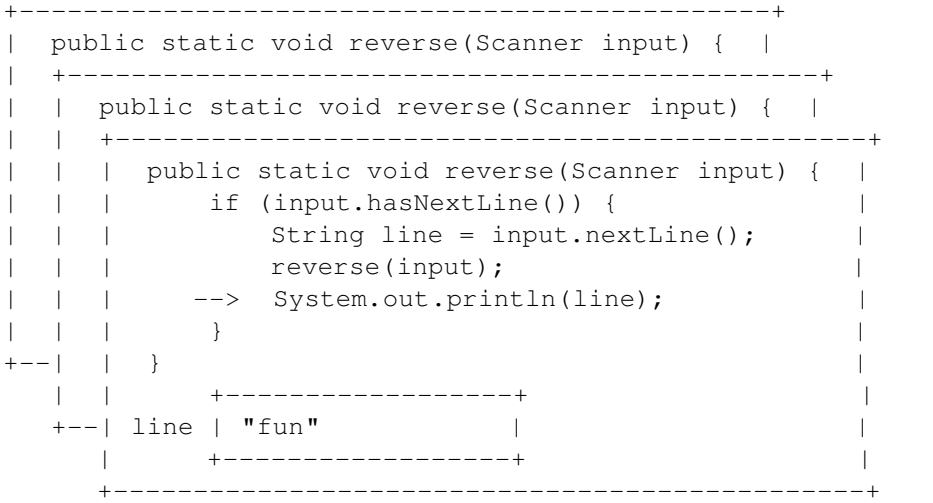


This one turns out to have the easy task, like our final robot that was asked to print a line of 0 stars. This time around the Scanner is empty (input.hasNextLine() returns false). This is our very important base case that stops this process from going on indefinitely. This version of the method recognizes that there are no lines to reverse, so it simply terminates.

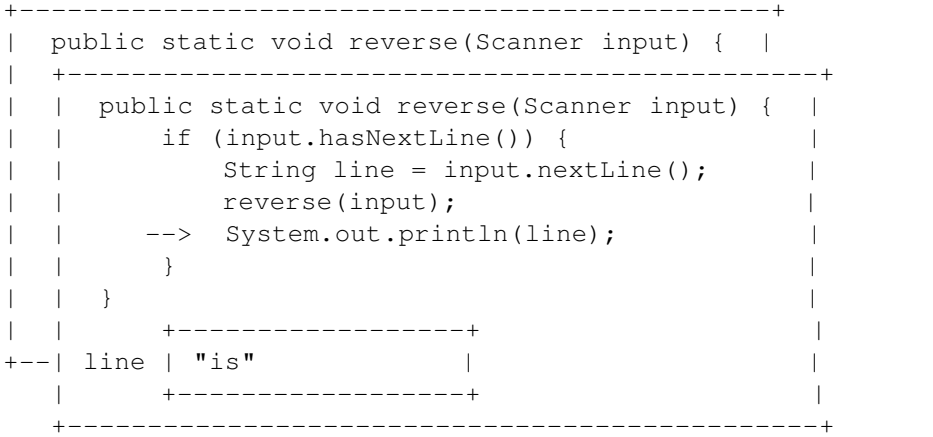
Then what? We're done with this call, so we throw it away and go back to where we were just before this call.



We've finished the call on reverse, so now we're positioned at the println right after it. So we print the text in our variable "line" (we print "no?") and terminate. And where does that leave us? This method goes away and we return to where we were just before:



So we print our line of text, which is "fun" and this version goes away:



So we execute this println for the text "is" and eliminate one more call:

```
+-----------------------------------------------+
|  public static void reverse(Scanner input) {  |
|      if (input.hasNextLine()) {               |
|          String line = input.nextLine();      |
|          reverse(input);                      |
|     -->  System.out.println(line);            |
|      }                                        |
|  }                                            |
|      +------------------+                     |
| line | "this"           |                     |
|      +------------------+                     |
+-----------------------------------------------+
```

Notice that we've written out three lines of text so far:

```
no?
fun
is
```

So our leap of faith was justified. The recursive call on reverse read in the three lines of text that came after the first line of input and printed them in reverse order. We complete the task by printing the first line of text, which leads to this overall output:

```
no?
fun
is
this
```

Then this version of the method terminates and we're done.

12.3 Recursive Functions

Both of the examples we have seen so far of recursion have been action oriented methods with a return type of void. In this section we will examine some of the issues that come up when you want to write methods that compute values and return a result.

Integer Exponentiation

Java provides a method Math.pow that allows you to compute an exponent. If you want to know what x^y is equal to, you can call Math.pow(x, y). Let's consider how we could implement the pow method. To keep things simple, we'll limit ourselves to the domain of integers. Because we are limiting ourselves to integers, we have to recognize an important precondition of our method. We won't be able to compute negative exponents because the results would not be integers.

So the method we want to write will look like this:

```
// pre : y >= 0
// post: returns x^y
public static int pow(int x, int y) {
    ...
}
```

We could obviously solve this problem by writing a loop, but we want to explore how to write the method recursively. Again the right place to start is to think about different cases. What would be the easiest exponent to compute? It's pretty easy to compute x^1, so that's a good candidate, but there is an even more basic case. The simplest possible exponent is 0. By definition, any integer to the 0 power is considered to be 1. So we can begin our solution as follows:

```
public static int pow(int x, int y) {
    if (y == 0) {
        // base case with y == 0
        return 1;
    } else {
        // recursive case with y > 0
        ...
    }
}
```

In the recursive case we know that y is greater than 0. In other words, there will be at least one factor of x to be included in the result. We know from mathematics that:

$$x^y = x * x^{y-1}$$

This equation expresses x to the y power in terms of x to a smaller power (y - 1). Therefore it can serve as our recursive case. All we have to is to translate it into its Java equivalent:

```
public static int pow(int x, int y) {
    if (y == 0) {
        // base case with y == 0
        return 1;
    } else {
        // recursive case with y > 0
        return x * pow(x, y - 1);
    }
}
```

This is a complete recursive solution. Tracing the execution of a recursive function is a little more difficult than with a void method because we have to keep track of the values being returned by each recursive call. The following is a trace of execution showing how we would compute 3^5:

```
pow(3, 5) = 3 * pow(3, 4)
 |     pow(3, 4) = 3 * pow(3, 3)
 |      |     pow(3, 3) = 3 * pow(3, 2)
 |      |      |     pow(3, 2) = 3 * pow(3, 1)
 |      |      |      |     pow(3, 1) = 3 * pow(3, 0)
 |      |      |      |      |     pow(3, 0) = 1
 |      |      |      |     pow(3, 1) = 3 * 1 = 3
 |      |      |     pow(3, 2) = 3 * 3 = 9
 |      |     pow(3, 3) = 3 * 9 = 27
 |     pow(3, 4) = 3 * 27 = 81
pow(3, 5) = 3 * 81 = 243
```

Notice that we make a series of six recursive calls in a row until we get to the base case of computing 3 to the 0 power. That call returns the value 1 and then the recursion unwinds, computing the various answers as it returns from each method call.

It is useful to think about what will happen if someone violates the precondition by asking for a negative exponent. For example, what if someone asks for pow(3, -1). The method would recursively ask for pow(3, -2), which would ask for pow(3, -3), which would ask for pow(3, -4) and so on. In other words, it leads to an infinite recursion. In some sense this is okay because someone calling the method should pay attention to the precondition. But it's not much work for us to handle this case in a more elegant manner. Our solution is structured as a series of cases, so we can simply add a new case for illegal exponents:

```
public static int pow(int x, int y) {
    if (y < 0) {
        throw new IllegalArgumentException("negative exponent");
    } else if (y == 0) {
        // base case with y == 0
        return 1;
    } else {
        // recursive case with y > 0
        return x * pow(x, y - 1);
    }
}
```

One of the advantages of writing functions recursively is that if we can identify other cases, we can potentially make the function more efficient. For example, suppose that you want to compute 2^{16}. In its current form, the method will multiply 2 by 2 by 2 a total of 16 times. We can do better than that. In particular, we know that if y is an even exponent, then:

$$x^y = (x^2)^{y/2}$$

So instead of computing 2^{16}, we can compute 4^8. This case can be added to our method in a relatively simple manner:

```
public static int pow(int x, int y) {
    if (y < 0) {
        throw new IllegalArgumentException("negative exponent");
    } else if (y == 0) {
        // base case with y == 0
        return 1;
    } else if (y % 2 == 0) {
        // recursive case with y > 0, y even
        return pow(x * x, y / 2);
    } else {
        // recursive case with y > 0, y odd
        return x * pow(x, y - 1);
    }
}
```

This version of the method is more efficient than the original. The following is a trace of execution for computing 2^{16}:

```
pow(2, 16) = pow(4, 8)
 |     pow(4, 8) = pow(16, 4)
 |      |     pow(16, 4) = pow(256, 2)
 |      |      |      pow(256, 2) = pow(65536, 1)
 |      |      |       |       pow(65536, 1) = 65536 * pow(65536, 0)
 |      |      |       |         |         pow(65536, 0) = 1
 |      |      |       |       pow(65536, 1) = 65536 * 1 = 65536
 |      |      |      pow(256, 2) = 65536
 |      |     pow(16, 4) = 65536
 |     pow(4, 8) = 65536
pow(2, 16) = 65536
```

Greatest Common Divisor

In Mathematics we often want to know the largest integer that goes evenly into two different integers, which is known as the greatest common divisor or gcd of the two integers. Let's explore how to write the gcd method recursively.

For now let's not worry about negative values of x and y. So we want to write this method:

```
// pre : x >= 0, y >= 0
// post: returns the greatest common divisor of x and y
public static int gcd(int x, int y) {
    ...
}
```

To introduce some variety, let's try to figure out the recursive case first and then figure out the base case. Suppose, for example, that we are asked to compute the gcd of 20 and 132. The answer is 4 because 4 goes evenly into each number and it is the largest integer that goes evenly into both.

There are many ways to compute the gcd of two numbers. One of the most efficient algorithms dates back at least to the time of Euclid and perhaps even farther. The idea is to eliminate any multiples of the smaller integer from the larger integer. For our example of 20 and 132, we know that:

```
132 = 20 * 6 + 12
```

There are 6 multiples of 20 in 132 with a remainder of 12. Euclid's algorithm says that we can ignore the 6 multiples of 20 and just focus on the value 12. In other words, we know that we can replace 132 with 12:

```
gcd(132, 20) = gcd(12, 20)
```

The proof of this principle is beyond the scope of this book, but that is the basic idea. This is easy to express in Java terms because the mod operator gives us the remainder when one number is divided by another. Expressing this principle in general terms, we know that:

```
gcd(x, y) = gcd(x % y, y) when y > 0
```

Again, the proof is beyond the scope of this book, but given this basic principle, we can produce a recursive solution to the problem. So we might try to write the method as follows:

```
public static int gcd(int x, int y) {
    if (...) {
        // base case
        ...
    } else {
        // recursive case
        return gcd(x % y, y);
    }
}
```

This isn't a bad first attempt, but it has a problem. It's not enough to be mathematically correct. We have to know that our recursive solution keeps reducing the overall problem to a simpler problem. If we start with 132 and 20, the method makes progress on the first call, but then it starts repeating itself:

```
gcd(132, 20) = gcd(12, 20)
        gcd(12, 20) = gcd(12, 20)
                gcd(12, 20) = gcd(12, 20)
                        gcd(12, 20) = gcd(12, 20)
                                ...
```

This will lead to infinite recursion. The Euclidean trick helped the first time around because for the first call x is greater than y (132 is greater than 20). So the algorithm makes progress only if the first number is larger than the second number.

The line of code that is causing us problems is this one:

```
return gcd(x % y, y);
```

When we compute (x % y), we are guaranteed to get a result that is smaller than y. That means that on the recursive call, the first value will always be smaller than the second value. To make the algorithm work, we need the opposite to be true. And we can make the opposite be true simply by reversing the order of the arguments:

```
return gcd(y, x % y);
```

On this call we are guaranteed to have a first value that is larger than the second value. If we trace this version of the method for computing the gcd of 132 and 20, we get the following:

```
gcd(132, 20) = gcd(20, 12)
         gcd(20, 12) = gcd(12, 8)
                 gcd(12, 8) = gcd(8, 4)
                         gcd(8, 4) = gcd(4, 0)
                                 ...
```

At this point we have to decide what the gcd of 4 and 0 is. It may seem strange, but the answer is that the gcd of 4 and 0 is 4. In general, gcd(n, 0) is always n. Obviously the gcd couldn't be any larger than n and n goes evenly into n. But n also goes evenly into 0 because 0 can be written as an even multiple of n (0 * n).

This observation leads us to the base case. If y is 0, then the gcd is x:

```
public static int gcd(int x, int y) {
    if (y == 0) {
        // base case with y == 0
        return x;
    } else {
        // recursive case with y > 0
        return gcd(y, x % y);
    }
}
```

With this base case we also solve the potential problem that the Euclidean formula depends on y not being 0. We still have to think about the case where one or both of x and y is negative. We could keep the precondition and throw an exception when one or both of the values is negative. It is more common in Mathematics to return the gcd of the absolute value of the two values. We can accomplish this with one extra case for negatives:

```
public static int gcd(int x, int y) {
    if (x < 0 || y < 0) {
        // recursive case with negative value(s)
        return gcd(Math.abs(x), Math.abs(y));
    } else if (y == 0) {
        // base case with y == 0
        return x;
    } else {
        // recursive case with y > 0
        return gcd(y, x % y);
    }
}
```

Common Programming Error: Infinite Recursion

Everyone who uses recursion to write programs eventually accidentally writes a solution that leads to infinite recursion. For example, below is a slight variation of the gcd method that doesn't work.

```
public static int gcd(int x, int y) {
    if (x <= 0 || y <= 0) {
        // recursive case with negative value(s)
        return gcd(Math.abs(x), Math.abs(y));
    } else if (y == 0) {
        // base case with y == 0
        return x;
    } else {
        // recursive case with y > 0
        return gcd(y, x % y);
    }
}
```

This solution is just slightly different than the one we wrote. In the test for negative values, this code tests whether x and y are less-than-or-equal to 0. The original code tests whether they are strictly less than 0. It doesn't seem like this should make much of a difference, but it does. If we execute this version of the code on our original problem of finding the gcd of 132 and 20, the program produces many lines of output that look like this:

```
        at Bug.gcd(Bug.java:9)
        at Bug.gcd(Bug.java:9)
        at Bug.gcd(Bug.java:9)
        at Bug.gcd(Bug.java:9)
        at Bug.gcd(Bug.java:9)
        at Bug.gcd(Bug.java:9)
        at Bug.gcd(Bug.java:9)
```

The first time you see this, you are likely to think that something has broken on your computer because you will get so many lines of output. The number of lines of output will vary from one system to another, but it's likely to be hundreds of lines of output and in some cases, thousands of lines of output. If you scroll all the way back up, you'll see that it begins with this message:

```
Exception in thread "main" java.lang.StackOverflowError
        at Bug.gcd(Bug.java:9)
        at Bug.gcd(Bug.java:9)
        at Bug.gcd(Bug.java:9)
        ...
```

Java is letting you know that the call stack has gotten too big. Why is this happening? Remember the trace of execution for this case:

```
gcd(132, 20) = gcd(20, 12)
         gcd(20, 12) = gcd(12, 8)
                 gcd(12, 8) = gcd(8, 4)
                         gcd(8, 4) = gcd(4, 0)
                                 ...
```

Think of what happens at this point when we call gcd(4, 0). The value of y is 0, which is our base case. So normally we would expect the method to return the value 4 and terminate. But the method begins by checking whether either of x or y is less-than-or-equal to 0. Since y is 0, this test evaluates to true. So we make a recursive call with the absolute value of x and y. But the absolute values of 4 and 0 are 4 and 0. In other words, we decide that the gcd(4, 0) must be equal to the gcd(4, 0), which must be equal to the gcd(4, 0):

```
gcd(132, 20) = gcd(20, 12)
         gcd(20, 12) = gcd(12, 8)
                 gcd(12, 8) = gcd(8, 4)
                         gcd(8, 4) = gcd(4, 0)
                                 gcd(4, 0) = gcd(4, 0)
                                         gcd(4, 0) = gcd(4, 0)
                                                 gcd(4, 0) = gcd(4, 0)
                                                         gcd(4, 0) = gcd(4, 0)
                                                         ...
```

In other words, this version generates infinitely many recursive calls. Think in terms of the pieces of paper that are stacked on top of each other as methods are called. Java allows you to make a lot of recursive calls, but eventually it runs out of space. When it does, it gives you a backtrace to let you know how you got to the error. In this case, the backtrace is not nearly as helpful as usual because almost all of the calls will involve the infinite recursion.

To handle these situations you have to look closely at the line number to see which line of your program generated the infinite recursion. In this case, we'd know that it is the recursive call for negative x and y values. That alone might be enough to allow us to pinpoint the error. If not, you might need to include println statements to figure out what is going on. For example, in this code, we could add a println just before the recursive call:

```
public static int gcd(int x, int y) {
    if (x <= 0 || y <= 0) {
        // recursive case with negative value(s)
        System.out.println("x = " + x + " and y = " + y);
        return gcd(Math.abs(x), Math.abs(y));
    } else if (y == 0) {
    ...
}
```

With that println in place, the code produces hundreds of lines of output of the form:

```
x = 4 and y = 0
```

If we examine that case closely, we'd see that we don't have negative values and we'd realize that we have to fix the test we are using.

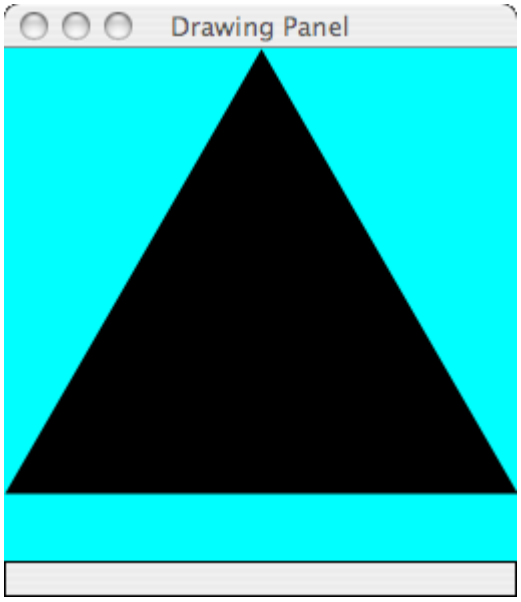
12.4 Recursive Graphics (optional)

There has been a great deal of interest in the past 30 years about an emerging field of Mathematics called *fractal geometry*. A fractal is a geometric object that is recursively constructed or self-similar. A fractal shape contains smaller versions of itself so that it looks similar at all magnifications.

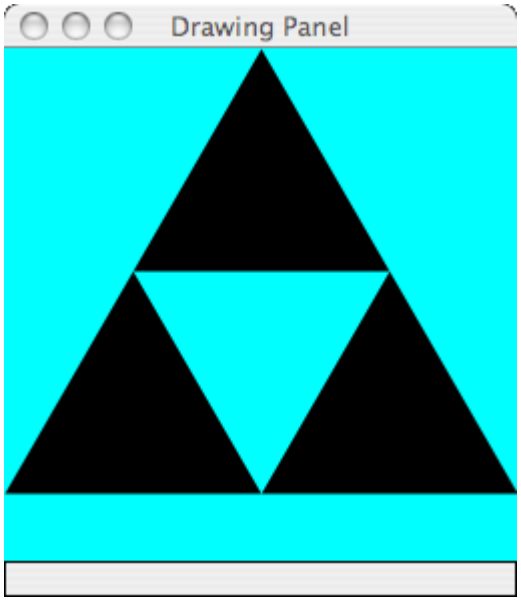
Benoit Mandelbrot created the fractal field in 1975 with his first publication about fractals in general and about a specific fractal that has come to be known as the Mandelbrot set. The most impressive aspect of fractal geometry is that extremely intricate and complex phenomena can be described with a simple set of rules. When Mandelbrot and others began drawing pictures of their fractals, they were an instant hit.

Many fractals can be described easily with recursion. As an example, we will explore a recursive method for drawing what is known as the Sierpinski triangle. We can't draw the actual fractal because it is composed of infinitely many subtriangles. Instead, we will write a method that produces various levels that approximate the actual fractal.

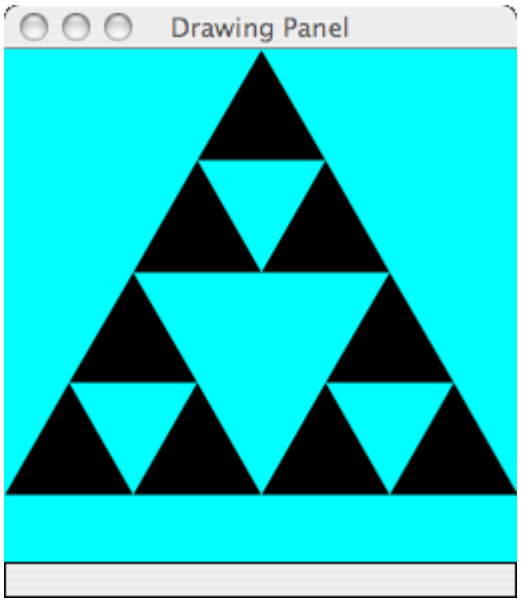
At level 1 we draw an equilateral triangle:



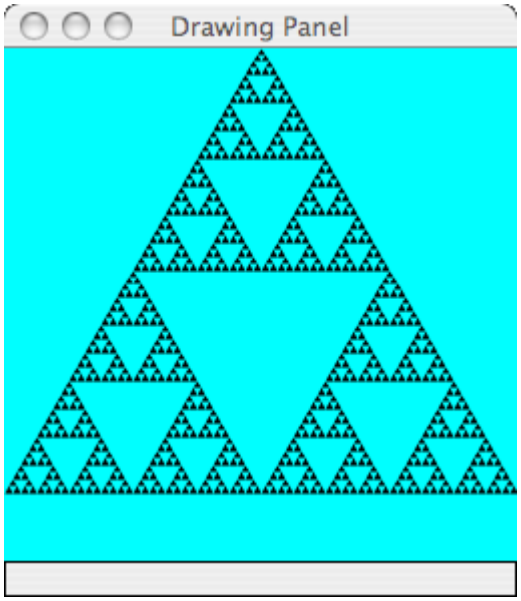
In going to level 2, we draw three smaller triangles that are contained within the original triangle:



We apply this principle in a recursive manner. Just as we replaced the original triangle with three inner triangles, we replace each of these three triangles with three inner triangles to obtain a figure with nine triangles to get to level 3:



This process continues indefinitely, making a more intricate pattern at each new level. At level 7 we get the following:



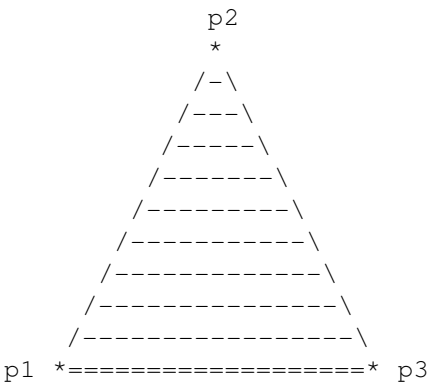
We can use the DrawingPanel class from Chapter 3 to solve this problem. We can pass the Graphics object for the panel to the method that is to draw the triangles. The method will also need to know the level to use and will need to know the three vertices of the triangle, which we can pass as point objects. So our method will look like this:

```
public static void drawFigure(int level, Graphics g,
                              Point p1, Point p2, Point p3) {
    ...
}
```

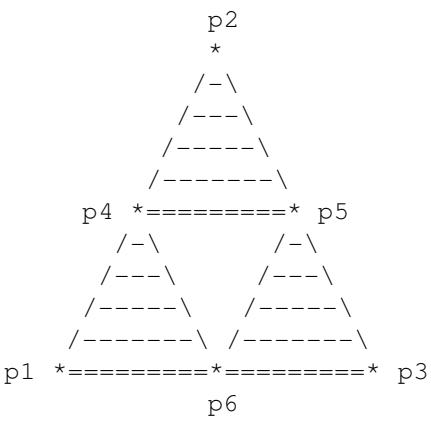
Our base case will be to draw the basic triangle for level 1. The Graphics class has methods for filling rectangles and ovals, but not for filling triangles. Fortunately there is a Polygon class in the java.awt package. You construct a Polygon and add a series of points to it. The Graphics class has a method called fillPolygon that we can use to fill the polygon once we've specified the three vertices. So our base case will look like this:

```
public static void drawFigure(int level, Graphics g,
                              Point p1, Point p2, Point p3) {
    if (level == 1) {
        // base case: simple triangle
        Polygon p = new Polygon();
        p.addPoint(p1.x, p1.y);
        p.addPoint(p2.x, p2.y);
        p.addPoint(p3.x, p3.y);
        g.fillPolygon(p);
    } else {
        // recursive case, split into 3 triangles
        ...
    }
}
```

Most of the work happens in the recursive case. We have to split the triangle into three smaller triangles. If we label the vertices of the overall triangle as follows:



We need to compute three new points that are the midpoints of the three sides of these triangles:



There are three different midpoint computations involved here, so it seems clear that it would be helpful to first write a method that will compute the midpoint of a segment given two endpoints:

```
public static Point midpoint(Point p1, Point p2) {
    return new Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2);
}
```

Given this method, we can easily compute the three new points. Looking at the diagram above, we see that:

- p4 is the midpoint of p1 and p2
- p5 is the midpoint of p2 and p3
- p6 is the midpoint of p1 and p3

Once we have computed those points, we can describe the smaller triangles as follows:

- in the lower-left is the triangle formed by p1, p4 and p6
- on top is the triangle formed by p4, p2 and p5
- in the lower-right is the triangle formed by p6, p5 and p3

The final detail we have to think about is the level. If you look again at the level 2 version of the figure, you will notice that it is composed of three simple triangles. In other words, the level 2 figure is composed of three level 1 figures. Similarly, the level 3 figure is composed of three level 2 figures which in turn are each composed of three level 1 figures. In general, if we have been asked to draw a level n figure, we do so by drawing three level (n - 1) figures.

By turning these observations into code, we can complete the recursive method:

```
public static void drawFigure(int level, Graphics g,
                              Point p1, Point p2, Point p3) {
    if (level == 1) {
        // base case: simple triangle
        Polygon p = new Polygon();
        p.addPoint(p1.x, p1.y);
        p.addPoint(p2.x, p2.y);
        p.addPoint(p3.x, p3.y);
        g.fillPolygon(p);
    } else {
        // recursive case, split into 3 triangles
        Point p4 = midpoint(p1, p2);
        Point p5 = midpoint(p2, p3);
        Point p6 = midpoint(p1, p3);

        // recurse on 3 triangular areas
        drawFigure(level - 1, g, p1, p4, p6);
        drawFigure(level - 1, g, p4, p2, p5);
        drawFigure(level - 1, g, p6, p5, p3);
    }
}
```

There is a limit to how many levels deep we can go. The DrawingPanel has a finite resolution, so at some point we won't be able to subdivide our triangles any further. Also, at each new level the number of triangles that we draw triples, which means that the number of triangles increases exponentially with the level.

Below is a complete program that allows the user to decide what level to use in drawing the figure.

```
1  import java.awt.*;
2  import java.util.*;
3
4  public class Sierpinski {
5      public static final int SIZE = 256;
6
7      public static void main(String[] args) {
8          // prompt for level
9          Scanner console = new Scanner(System.in);
10         System.out.print("What level do you want? ");
11         int level = console.nextInt();
12
13         // initialize drawing panel
14         DrawingPanel p = new DrawingPanel(SIZE, SIZE);
15         p.setBackground(Color.CYAN);
16         Graphics g = p.getGraphics();
17
18         // compute triangle endpoints and begin recursion
19         int triangleHeight = (int) Math.round(SIZE * Math.sqrt(3.0) / 2.0);
20         Point p1 = new Point(0, triangleHeight);
21         Point p2 = new Point(SIZE / 2, 0);
22         Point p3 = new Point(SIZE, triangleHeight);
23         drawFigure(level, g, p1, p2, p3);
24     }
25
26     // Draws a Sierpinski fractal to the given level inside the triangle
27     // whose vertices are (p1, p2, p3).
28     public static void drawFigure(int level, Graphics g,
29                                   Point p1, Point p2, Point p3) {
30         if (level == 1) {
31             // base case: simple triangle
32             Polygon p = new Polygon();
33             p.addPoint(p1.x, p1.y);
34             p.addPoint(p2.x, p2.y);
35             p.addPoint(p3.x, p3.y);
36             g.fillPolygon(p);
37         } else {
38             // recursive case, split into 3 triangles
39             Point p4 = midpoint(p1, p2);
40             Point p5 = midpoint(p2, p3);
41             Point p6 = midpoint(p1, p3);
42
43             // recurse on 3 triangular areas
44             drawFigure(level - 1, g, p1, p4, p6);
45             drawFigure(level - 1, g, p4, p2, p5);
46             drawFigure(level - 1, g, p6, p5, p3);
47         }
48     }
49
50     // returns the midpoint of p1 and p2
51     public static Point midpoint(Point p1, Point p2) {
52         return new Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2);
53     }
54 }
```

12.5 Case Study: Prefix Evaluator

In this section we will explore the use of recursion to evaluate complex numeric expressions. We will begin by exploring the different conventions for specifying numeric expressions and then we will see how recursion makes it relatively easy to implement one of the standard conventions.

Infix, Prefix and Postfix Notation

When we write numeric expressions in a Java program we put numeric operators like "+" and "*" between the two operands, as in:

```
3.5 + 8.2
9.1 * 12.7
7.8 * (2.3 + 2.5)
```

Putting the operator between the operands is a convention known as *infix notation*. A second convention is to put the operator in front of the two operands, as in:

```
+ 3.5 8.2
* 9.1 12.7
* 7.8 + 2.3 2.5
```

Putting the operator in front of the operands is a convention known as *prefix notation*. Prefix notation looks odd for symbols like "+" and "*", but it is more like mathematical function notation where the name of the function goes first. For example, if instead of using the operators we instead were calling methods, we would write:

```
plus(3.5, 8.2)
times(9.1, 12.7)
times(7.8, plus(2.3, 2.5))
```

There is a third convention known as *postfix notation* where the operator appears after the two operands, as in:

```
3.5 8.2 +
9.1 12.7 *
7.8 2.3 2.5 + *
```

Postfix notation is also known as Reverse Polish Notation or RPN. For many years Hewlett-Packard has sold scientific calculators that use RPN rather than normal infix notation.

We are so used to infix notation that it takes a while to learn the other two conventions. One of the interesting facts you will discover if you take the time to learn the other conventions is that infix is the only notation that requires parentheses. The other two notations are unambiguous.

Below is a table that summarizes the three notations.

Arithmetic Notations

Notation	Description	Examples
infix	operator between operands	2.3 + 4.7 2.6 * 3.7 (3.4 + 7.9) * 18.6 + 2.3 / 4.7
prefix	operator before operands (functional notation)	+ 2.3 4.7 * 2.6 3.7 + * + 3.4 7.9 18.6 / 2.3 4.7
postfix	operator after operands (Reverse Polish Notation)	2.3 4.7 + 2.6 3.7 * 3.4 7.9 + 18.6 * 2.3 4.7 / +

Prefix Evaluator

Of the three standard notations, prefix notation is the one most easily implemented with recursion. In this section we will write a method that reads a prefix expression from a Scanner and that computes its value. So our method should look like this:

```
// pre : input contains a legal prefix expression
// post: expression is consumed and the result is returned
public static double evaluate(Scanner input) {
    ...
}
```

Before we can begin we have to consider what kind of input we are going to get. As the precondition indicates, we will assume that the Scanner contains a legal prefix expression. The simplest possible expression would be a number, as in:

```
38.9
```

There isn't much of an expression to evaluate in this case. We can simply read and return the number. More complex prefix expressions will involve one or more operators. Remember that the operator goes in front of the operands in a prefix expression. So a slightly more complex expression would be to have two numbers as operands with an operator in front, as in:

```
+ 2.6 3.7
```

This expression could itself be an operand in a larger expression. For example, we might ask for:

```
* + 2.6 3.7 + 5.2 18.7
```

At the outermost level we have a multiplication operator with two operands:

```
   *      + 2.6 3.7   + 5.2 18.7
   |      \-------/   \--------/
operator  operand #1  operand #2
```

In other words, this expression is computing the product of two sums. Here is the same expression in the more familiar infix notation:

```
(2.6 + 3.7) * (5.2 + 18.7)
```

These expressions can become arbitrarily complex. But the key observation to make about the expressions that involve operators is that they all begin with an operator. In other words, every prefix expression is of one of two forms:

- a simple number
- an operator followed by two operands

This observation will become a roadmap for our recursive solution. The simplest prefix expression will be a number and we can distinguish it from the other case because the other expressions begin with an operator. So we can begin our recursive solution by looking to see if the next token in the Scanner is a number. If so, then we have a simple case and we can simply read and return the number:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        ...
    }
}
```

Turning our attention to the recursive case, we know that the input must be composed of an operator followed by two operands. We can begin by reading the operator:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        String operator = input.next();
        ...
    }
}
```

At this point we reach a critical decision. We have read in the operator. Now we need to somehow read in the first operand and after that we have to read in the second operand. If we knew that the operands were simple numbers, we could say:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        String operator = input.next();
        double operand1 = input.nextDouble();
        double operand2 = input.nextDouble();
        ...
    }
}
```

But we have no guarantee that the operands are simple numbers. They might be complex expressions that begin with operators. Your instinct might be to test whether or not the original operator is followed by another operator (in other words, whether the first operand begins with an operator), but that reasoning won't lead you to a satisfactory outcome. Remember that the expressions can be arbitrarily complex. So there might be dozens of operators to be processed in either of the operands.

The solution to this puzzle involves recursion. We have two operands that we need to read from the Scanner and they might be very complex. But we know that they are in prefix form and we know that they aren't as complex as the original expression we were asked to evaluate. The key is to recursively evaluate each of the two operands:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        String operator = input.next();
        double operand1 = evaluate(input);
        double operand2 = evaluate(input);
        ...
    }
}
```

The solution is so simple that it almost doesn't seem fair to solve the problem so easily. But this actually works. Of course, we still have the task of evaluating the operator. So after the two recursive calls we will have an operator and two numbers (say, "+" and 3.4 and 2.6). It would be nice if we could just say:

```
return operand1 operator operand2;
```

Unfortunately, Java doesn't work that way. We have to use a nested if/else statement to test what kind of operator we have and to return an appropriate value, as in:

```
if (operator.equals("+")) {
    return operand1 + operand2;
} else if (operator.equals("-")) {
    return operand1 - operand2;
} else if (operator.equals("*")) {
    ...
```

This code can be included in its own method so that our recursive method can stay fairly short:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        String operator = input.next();
        double operand1 = evaluate(input);
        double operand2 = evaluate(input);
        return evaluate(operator, operand1, operand2);
    }
}
```

Complete Program

When you program with recursion you'll notice two things. First, the recursive code that you write tends to be fairly short even though it might be solving a very complex task. Second, you will generally find that most of your program ends up being supporting code for the recursion that does low-level tasks. For this problem we have a short and powerful prefix evaluator, but we need to include some supporting code that explains the program to the user, that prompts for a prefix expression and that reports the result. We also found that we needed a method that would evaluate an operator and two operands. The nonrecursive parts of the program are fairly straightforward, so they are included below without detailed discussion.

```
 1  import java.util.*;
 2
 3  public class PrefixEvaluator {
 4      public static void main(String[] args) {
 5          Scanner console = new Scanner(System.in);
 6          System.out.println("This program evaluates prefix expressions");
 7          System.out.println("for operators +, -, *, / and %");
 8          System.out.print("expression? ");
 9          System.out.println("value = " + evaluate(console));
10      }
11
12      // pre : input contains a legal prefix expression
13      // post: expression is consumed and the result is returned
14      public static double evaluate(Scanner input) {
15          if (input.hasNextDouble()) {
16              return input.nextDouble();
17          } else {
18              String operator = input.next();
19              double operand1 = evaluate(input);
20              double operand2 = evaluate(input);
21              return evaluate(operator, operand1, operand2);
22          }
23      }
24
25      // pre : operator is one of +, -, *, / or %
26      // post: returns the result of applying the given operator to
27      //       the given operands
28      public static double evaluate(String operator, double operand1,
29                                    double operand2) {
30          if (operator.equals("+")) {
31              return operand1 + operand2;
32          } else if (operator.equals("-")) {
33              return operand1 - operand2;
```

```
34          } else if (operator.equals("*")) {
35              return operand1 * operand2;
36          } else if (operator.equals("/")) {
37              return operand1 / operand2;
38          } else if (operator.equals("%")) {
39              return operand1 % operand2;
40          } else {
41              throw new RuntimeException("illegal operator " + operator);
42          }
43      }
44  }
```

The program can handle simple numbers, as in the following sample execution:

```
This program evaluates prefix expressions
for operators +, -, *, / and %
expression? 38.9
value = 38.9
```

It can also handle expressions with a single operator, as in:

```
This program evaluates prefix expressions
for operators +, -, *, / and %
expression? + 2.6 3.7
value = 6.300000000000001
```

And it handles the case we considered that involved a product of two sums:

```
This program evaluates prefix expressions
for operators +, -, *, / and %
expression? * + 2.6 3.7 + 5.2 18.7
value = 150.57000000000002
```

It can handle arbitrarily complex expressions, as in the following sample execution:

```
This program evaluates prefix expressions
for operators +, -, *, / and %
expression? / + * - 17.4 8.9 - 3.9 4.7 18.4 - 3.8 * 7.9 2.3
value = -0.8072372999304106
```

The expression being computed in this example is the prefix equivalent of the following infix expression:

```
((17.4 - 8.9) * (3.9 - 4.7) + 18.4) / (3.8 - 7.9 * 2.3)
```

Chapter Summary

- Recursion is an algorithmic technique where a method calls itself. A method that uses recursion is called a recursive method.
- Recursive methods contain two cases: A base case that the method can solve directly without recursion, and a recursive case where the method reduces a problem into a simpler problem of the same kind with a recursive call.

- Recursive method calls work internally by storing information about each call into a structure called a call stack. When the method calls itself, the information about this call and its parameter information is placed on top of the stack. When a method call finishes executing, its information is removed from the top of the stack and the program returns to the method call underneath.

- Recursion is useful for representing many common mathematical functions, such as exponentiation and finding the greatest common divisor (GCD) of two integers. Recursive solutions for math operations often match the mathematical definitions for those operations more closely than an iterative solution.

- A recursive method without a base case, or one where the recursive case doesn't properly transition into the base case, can lead to infinite recursion. Infinite recursion crashes your program because the method calls itself repeatedly until Java runs out of call stack memory.

- Recursion can be used to draw graphical figures in complex patterns, including fractal images. Fractals are images that are recursively constructed or self-similar, that is, a shape that appears similar at all scales of magnification and is therefore often referred to as "infinitely complex."

Self-Check Problems

Section 12.1: Thinking Recursively

1. What is recursion? How does a recursive method differ from a standard iterative method?

2. What are base cases and recursive cases? Why does a recursive method need to have both?

3. Consider the following method:

```
public static void mystery2(int n) {
    if (n <= 1) {
        System.out.print(n);
    } else {
        mystery2(n / 2);
        System.out.print(", " + n);
    }
}
```

   For each call below, indicate what output is produced by the method:

   - mystery2(1)
   - mystery2(2)
   - mystery2(3)
   - mystery2(4)
   - mystery2(16)
   - mystery2(30)
   - mystery2(100)

4. Convert the following iterative method into a recursive method:

```
// Writes n characters in a pattern, such as:
// writeChars(1) prints *
// writeChars(2) prints **
// writeChars(3) prints <*>
// writeChars(4) prints <**>
// writeChars(6) prints <<**>>
// writeChars(12) prints <<<<<**>>>>>
public static void writeChars(int n) {
    int half = (n - 1) / 2;
    for (int i = 0; i < half; i++) {
        System.out.print("<");
    }

    if (n % 2 == 0) {
        System.out.print("**");
    } else {
        System.out.print("*");
    }

    for (int i = 0; i < half; i++) {
        System.out.print(">");
    }
}
```

5. Convert the following iterative method into a recursive method:

```
// Prints each character of the string reversed twice.
// stutterReverse("hello") prints oolllleehh
public static void stutterReverse(String s) {
    for (int i = s.length() - 1; i >= 0; i--) {
        System.out.print(s.charAt(i));
        System.out.print(s.charAt(i));
    }
}
```

Section 12.2: A Better Example of Recursion

6. What is a call stack, and how does it relate to recursion?

7. What would be the effect if the code for the reverse method were changed to the following:

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        String line = input.nextLine();
        System.out.println(line);   // swapped order
        reverse(input);             // swapped order
    }
}
```

8. What would be the effect if the code for the reverse method were changed to the following:

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        reverse(input);             // moved this line
        String line = input.nextLine();
        System.out.println(line);
    }
}
```

Section 12.3: Recursive Functions

9. What are the differences between the two versions of the pow method shown? What advantage does the second version have over the first version? Are both versions recursive?

10. Convert the following iterative method into a recursive method:

```
// Returns n!, such as 5! = 1*2*3*4*5
public static int factorial(int n) {
    int product = 1;
    for (int i = 1; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

11. Consider the following method:

```
public static int mystery1(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return mystery1(x - y, y);
    }
}
```

    For each call below, indicate what value is returned:

    - mystery1(6, 13)
    - mystery1(14, 10)
    - mystery1(37, 10)
    - mystery1(8, 2)
    - mystery1(50, 7)

12. The following method has a bug that leads to infinite recursion. What correction fixes the code?

```
// Adds the digits of the given number.
// Example: digitSum(3456) returns 3+4+5+6 = 18
public static int digitSum(int n) {
    if (n > 10) {
        // base case (small number)
        return n;
    } else {
        // recursive case (large number)
        return n % 10 + digitSum(n / 10);
    }
}
```

Section 12.4: Recursive Graphics (optional)

13. What is a fractal image? How does recursive programming help to draw fractals?

14. Write Java code to create and draw a Polygon in the shape of a regular hexagon.

Exercises

1. Write a recursive method named starString that accepts an integer as a parameter and prints a string of stars (asterisks) 2^n long (i.e., 2 to the nth power). For example:

```
starString(0) should print * (because 2^0 == 1)
starString(1) should print ** (because 2^1 == 2)
starString(2) should print **** (because 2^2 == 4)
starString(3) should print ******** (because 2^3 == 8)
starString(4) should print **************** (because 2^4 == 16)
```

   The method should throw an IllegalArgumentException if passed a value less than 0.

2. Write a method named writeNums that takes an integer n as a parameter and that prints the first n integers starting with 1 to the console in sequential order and separated by commas. For example, the following calls:

```
writeNums(5);
System.out.println(); // to complete the line of output
writeNums(12);
System.out.println(); // to complete the line of output
```

   should produce the output:

```
1, 2, 3, 4, 5
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

   Your method should throw an IllegalArgumentException if passed a value less than 1.

3. Write a method writeSequence that accepts an integer n as a parameter and that prints to the console a symmetric sequence of n numbers with descending integers ending in 1 followed by ascending integers beginning with 1. The table below indicates the output that should be produced for various values of n:

```
Method Call               Output Produced
-----------------------------------------
writeSequence(1);         1
writeSequence(2);         1 1
writeSequence(3);         2 1 2
writeSequence(4);         2 1 1 2
writeSequence(5);         3 2 1 2 3
writeSequence(6);         3 2 1 1 2 3
writeSequence(7);         4 3 2 1 2 3 4
writeSequence(8);         4 3 2 1 1 2 3 4
writeSequence(9);         5 4 3 2 1 2 3 4 5
writeSequence(10);        5 4 3 2 1 1 2 3 4 5
```

Notice that for odd numbers the sequence has a single 1 in the middle while for even values it has two 1s in the middle. Your method should throw an IllegalArgumentException if passed a value less than 1.

4. Write a recursive method named stutter that accepts an integer as a parameter and returns the integer obtained by replacing every digit of n with two of that digit. For example, stutter(348) returns 334488. The call stutter(0) returns 0. Calling stutter on a negative number returns the negation of calling stutter on the corresponding positive number; for example, stutter(-789) returns -778899.

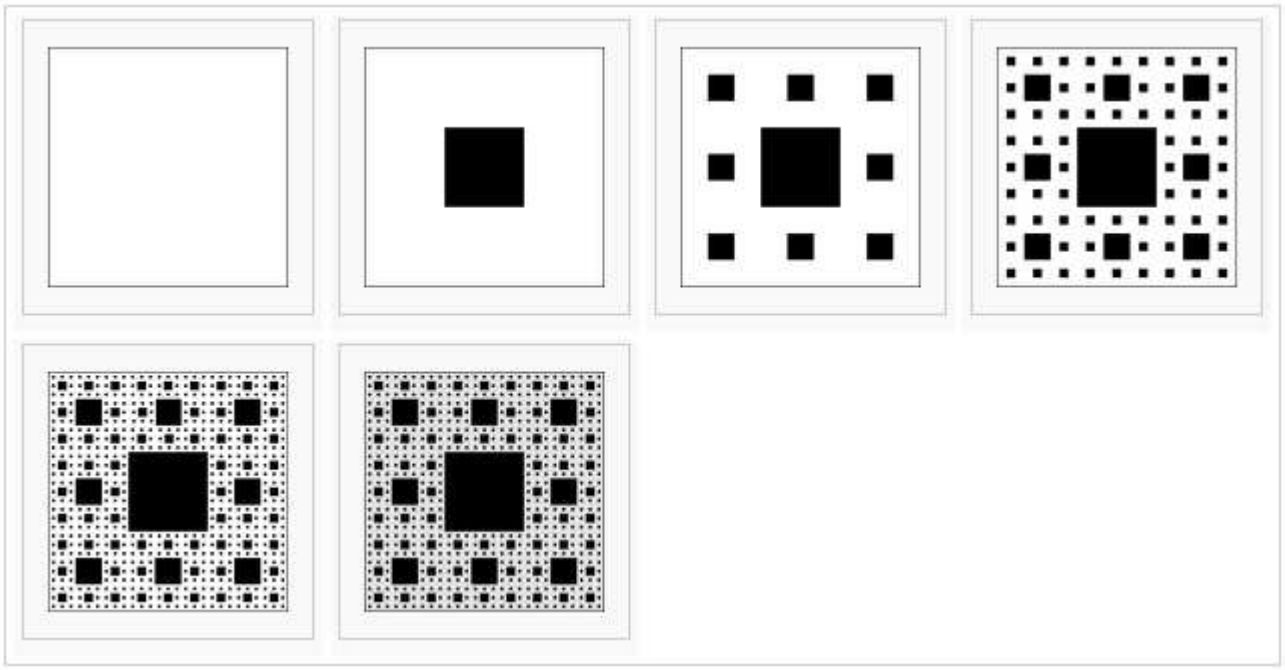
5. Write a recursive method named writeBinary that accepts an integer as a parameter and writes its binary representation to the console. For example, writeBinary(44) prints 101100.

6. Write a recursive method named permut that accepts two integers n and r as parameters and returns the number of unique permutations of r items from a group of n items. For given values of n and r, this value P(n, r) can be computed as follows:

$$P(n, r) = \frac{n!}{(n - r)!}.$$

For example, permut(7, 4) should return 840. It may be helpful to note that permut(6, 3) returns 120, or 840 / 7.

7. The Sierpinski carpet is a fractal defined as follows. The construction of the Sierpinski carpet begins with a square. The square is cut into 9 congruent subsquares in a 3-by-3 grid, with the central subsquare removed. The same process is then applied recursively to the 8 other subsquares. The following diagram shows the first few iterations of the carpet.



Write a program to draw the carpet on a DrawingPanel recursively.

8. The Cantor set is a fractal defined by repeatedly removing the middle thirds of line segments.

Write a program to draw the Cantor set on a DrawingPanel recursively.

Programming Projects

1. Write a recursive program to solve the "Missionaries and Cannibals" problem. Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross?

   Your output should include the initial problem, the moves you make, and a "picture" of the current state of the puzzle after each move. Your final output should produce only the moves and picture for those states that are on the "solution path."

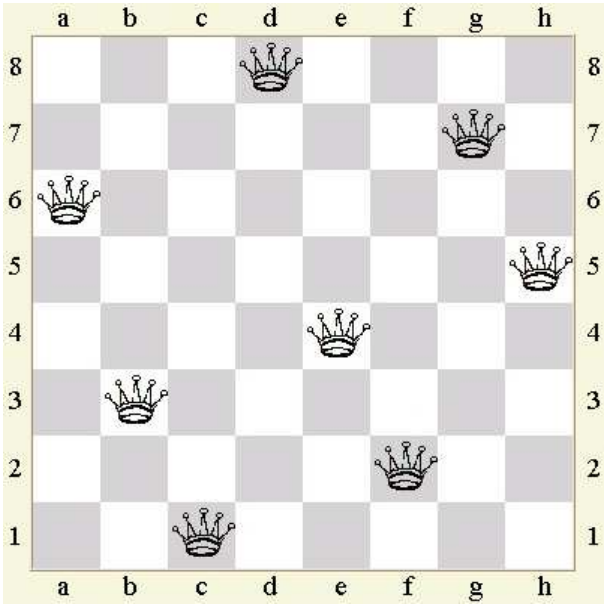
2. Write a recursive program to solve the Towers of Hanoi puzzle. The puzzle involves manipulating disks and three different towers where the disks can be placed. You are given a certain number of disks (4 in the example below) on one of the three towers. The disks have decreasing diameters, with the largest disk on the bottom:

```
          |                |                |
       *******             |                |
     ***********           |                |
   ***************         |                |
 *******************       |                |
          A                B                C
```

The object of the puzzle is to get all of the disks from one Tower to another, say, from A to B. The third tower is provided as temporary storage space as you move disks around. You are only allowed to move one disk at a time, and you are not allowed to place a disk on top of a smaller one (i.e., one with smaller diameter).

Examine the rather simple solutions for 1, 2, and 3 disks, and see if you can't discern a pattern. Then write a program that will solve the Towers of Hanoi puzzle for any number of disks. (Hint: Moving 4 disks is a lot like moving three disks, except that one additional disk is on top.)

3. Write a recursive program to solve the Eight Queens puzzle. The puzzle involves placing eight queens on a standard chess board in such a way that no two threaten each other. The following is one example solution to this problem:

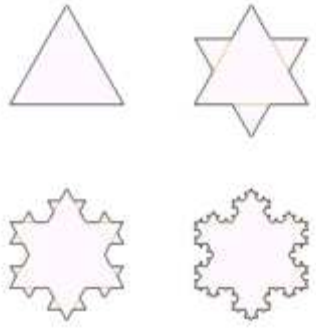


Your recursive program should explore all possible placements of eight queens on the board, reporting all configurations where no two queens are in jeapordy.

4. The Koch snowflake is a fractal created by starting with a line segment, then recursively altering each line segment as follows:

    1. divide the line segment into three segments of equal length.
    2. draw an equilateral triangle that has the middle segment from step 1 as its base.
    3. remove the line segment that is the base of the triangle from step 2.

    The following diagram shows the first several iterations of the snowflake.



Write a program to draw the Koch snowflake on a DrawingPanel recursively.

Stuart Reges
Marty Stepp

Chapter 13
Searching and Sorting

Copyright © 2006 by Stuart Reges and Marty Stepp

- 13.1 Searching and Sorting in the Java Class Libraries
  - Binary Search
  - Sorting
  - Shuffling
  - Custom Ordering with Comparators
- 13.2 Program Efficiency
  - Algorithm Runtimes
  - Complexity Classes
- 13.3 Implementing Searching Algorithms
  - Sequential Search
  - Binary Search
  - Searching Objects
- 13.4 Implementing Sorting Algorithms
  - Selection Sort
  - Merge Sort
  - Other Sorting Algorithms

Introduction

When we deal with large amounts of data, we often want the ability to search the data for a particular value. For example, you might want to search the internet for a web page that contains a certain keyword, or you might wish to search a phone book for a person's phone number.

It's also a useful operation to be able to rearrange a collection of data into sorted order. For example, you may wish to sort a list of students' course grade data by name, by student ID, or by grade.

In this chapter we'll look at ways to use Java's class libraries to search and sort data. Afterward we'll learn how to implement some searching and sorting algorithms of our own, as well as talking more generally about how to observe and analyze the runtime of algorithms.

13.1 Searching and Sorting in the Java Class Libraries

If you have an ArrayList collection, you can search it by calling its indexOf method. This method examines each element of the list, looking for a target value. It returns the first index at which the target value occurs in the list, or -1 if the target doesn't occur.

Imagine a large ArrayList containing all words in a book.

```
// read book into ArrayList
Scanner in = new Scanner(new File("mobydick.txt"));
ArrayList<String> words = new ArrayList<String>();
while (in.hasNext()) {
    words.add(in.next());
}
return words;
```

You could use the indexOf method to see whether a given word appears in the book, and at what index.

```
// search list using indexOf
System.out.print("Your word? ");
Scanner console = new Scanner(System.in);
String word = console.nextLine();

int index = words.indexOf(word);
if (index >= 0) {
    System.out.println(word + " is word #" + index + " in the book.");
} else {
    System.out.println(word + " isn't found in the book.");
}
```

The indexOf method is what we call a *sequential search* because it examines each element of the list in sequence until it finds the one you're looking for. If it reaches the end of the list without finding your element, it returns -1. When searching a 1,000,000 element list for an element at index 675,000, a sequential search would have to examine all 675,000 elements up to that point before it found the right value.

If you have an array of data instead of a list, there's no prewritten method to sequentially search it. You'll have to write the code yourself (as we'll do later in this chapter), or put the array's elements into an ArrayList first and search that with indexOf.

Binary Search

Sometimes we want to search through elements of an array or list that we know is in sorted order. For example, maybe we want to know whether "queasy" is a real English word, so we search the contents of an alphabetized dictionary text file. Or perhaps you're looking for a book written by Robert Louis Stevenson in a stack of books sorted by the author's last name. If the dictionary is large or the stack of books is tall, it's unlikely that you'd want to sequentially examine all of them.

There's a better algorithm named *binary search* that searches sorted data much faster than a sequential search. A normal sequential search of a million-element array may have to examine all the elements, but a binary search will only need to look at around 20 of them. Java's class libraries contain methods that implement the binary search algorithm for arrays and lists.

Binary search begins by examining the center element of the array. If the center element is smaller than the target we're searching for, there's no reason to examine any elements to the left of the center (ones at lower indexes). Conversely, if the center element is larger than the target we're searching for, there's no reason to examine any elements to the right of the center (ones at greater indexes). On each pass of the algorithm, half the search space is eliminated from consideration, so the target value is found much faster than in a sequential search.

Consider the example of a guessing game where the computer thinks of a number between 1 and 100 and the user tries to guess it. After each incorrect guess, the programs gives the user a hint about whether its guess was too high or too low. A poor algorithm for this game is to guess 1, 2, 3, and so on. A smarter algorithm is to guess the middle number and cut the range in half each time based on whether the guess was too high or too low.

Passes of a binary search for a number between 1 and 100.

```
0    10   20   30   40   50   60   70   80   90   100
guess = 50              ^
incorrect.  hint: higher

0    10   20   30   40   50   60   70   80   90   100
guess = 75                        ^
incorrect.  hint: lower

0    10   20   30   40   50   60   70   80   90   100
guess = 62                     ^
incorrect.  hint: higher

0    10   20   30   40   50   60   70   80   90   100
guess = 69                          ^
incorrect.  hint: higher

0    10   20   30   40   50   60   70   80   90   100
guess = 72                            ^
incorrect.  hint: lower

0    10   20   30   40   50   60   70   80   90   100
guess = 71                            ^
correct!
```

Binary search uses this same idea when searching a sorted array for a target value. The algorithm scales amazingly well to large input data; when searching a one million element list, a binary search would have to examine only around 20 elements.

The Arrays class in the java.util package contains a static method named binarySearch that implements the binary search algorithm. It accepts an array of any suitable type and a target value as its parameters and returns the index where you can find the target element. If the element isn't found, a negative index is returned. Here's a quick example usage of this method:

```
// demonstrate the Arrays.binarySearch method
String[] strings = {"a", "b", "c", "d", "e", "f", "g", "h"};
int index = Arrays.binarySearch(strings, "e");   // 4
System.out.println("e is found at index " + index);
```

If you're using a list such as an ArrayList instead, you can call the static method Collections.binarySearch to search the list of elements.

```
String[] strings = {"a", "b", "c", "d", "e", "f", "g", "h"};
ArrayList<String> list = new ArrayList<String>();
for (String s : strings) {
    list.add(s);
}

// demonstrate the Collections.binarySearch method
int index = Collections.binarySearch(list, "e");   // 4
System.out.println("e is found at index " + index);
```

In the case of arrays or lists, the data must be in sorted order to use the binarySearch method, because it relies on the sortedness to more quickly find the target value. If you call binarySearch on unsorted data, the results are undefined and the algorithm doesn't promise that it will return the right answer.

The following program demonstrates a binary search for a word in a dictionary file.

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class FindWords {
 5      public static void main(String[] args) throws FileNotFoundException {
 6          Scanner in = new Scanner(new File("words.txt"));
 7          ArrayList<String> words = new ArrayList<String>();
 8          while (in.hasNext()) {
 9              words.add(in.next());
10          }
11
12          System.out.print("Your word? ");
13          Scanner console = new Scanner(System.in);
14          String word = console.nextLine().trim().toLowerCase();
15          int index = Collections.binarySearch(words, word);
16          if (index >= 0) {
17              System.out.println(word + " is word #" + index + " in the dictionary.");
18          } else {
19              System.out.println(word + " isn't found in the dictionary.");
20          }
21      }
22  }
```

Here are two example runs of the program and their resulting output:

```
Your word? tugboat
tugboat is word #159226 in the dictionary.

Your word? googlymoogly
googlymoogly isn't found in the dictionary.
```

Sorting

Sorting is a very common and important computing task. When browsing your hard drive, you might sort your files by filename, extension, and date. When playing music, you might sort your songs by artist, year, or genre. We also saw that sorted arrays and lists can be searched very quickly with the binary search algorithm.

Sorting functionality is provided by the Java class libraries for arrays and lists. An array can be sorted with the Arrays.sort method:

```
// demonstrate the Arrays.sort method
String[] strings = {"c", "b", "g", "h", "d", "f", "e", "a"};
Arrays.sort(strings);
System.out.println(Arrays.toString(strings));
```

The preceding code produces the following output:

```
[a, b, c, d, e, f, g, h]
```

The array must be of a type that can be compared, that is, one that implements the Comparable interface. For example, you can sort an array of ints or Strings, but you can't sort an array of Point objects or Color objects.

There is also a method named Collections.sort that accepts a list such as an ArrayList as a parameter and puts its elements into sorted order. The following code produces the same output:

```
// demonstrate the Collections.sort method
ArrayList<String> list = new ArrayList<String>();
for (String s : strings) {   // uses previous array
    list.add(s);
}

Collections.sort(list);
System.out.println(list);
```

When used with primitive data, the Arrays.sort method uses an algorithm named quicksort. Collections.sort and Arrays.sort use a different algorithm named merge sort when dealing with object data. We'll discuss implementation of merge sort in detail later in the chapter.

Shuffling

The opposite of sorting might be the task of shuffling data, or rearranging its elements into a random order. Why would one want to do this?

One case would be when you want a random permutation of a list of numbers. Say that you want a random permutation of the numbers from 1 through 5. You could make an array or list and put five random values into it, but you'd have to write some clumsy code to confirm that the random values were unique and all fell in the proper range. A better way to do it would be to just store the numbers 1 through 5 into a list and shuffle the list.

Another example is when implementing a card game. You might have a card deck stored as a list of Card objects. To shuffle the deck of cards, you'd like to rearrange the card objects into a random ordering.

There's a method named Collections.shuffle that accepts a list as its parameter and rearranges its elements randomly. Here's a quick code example that creates a deck of card Strings, shuffles it, then examines the card at the end of the deck.

```
String[] ranks = {"2", "3", "4", "5", "6", "7", "8", "9",
                  "10", "Jack", "Queen", "King", "Ace"};
String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};

ArrayList<String> deck = new ArrayList<String>();
for (String rank : ranks) {
    for (String suit : suits) {
        deck.add(rank + " of " + suit);
    }
}
Collections.shuffle(deck);
System.out.println("Top card = " + deck.get(deck.size() - 1));
```

The code randomly produces output such as the following, a different output on each run:

```
Top card = 10 of Spades
```

The following table briefly summarizes the useful static methods in Java's class libraries for searching, sorting, and shuffling:

Searching and Sorting in Java's Class Libraries

Method	Description
`Arrays.binarySearch(array)`	Returns the index of the given value in the given array, assuming that the array's elements are currently in sorted order
`Arrays.sort(array)`	Arranges the given array's elements into sorted order
`Collections.binarySearch(list)`	Returns the index of the given value in the given list, assuming that the list's elements are currently in sorted order
`Collections.shuffle(list)`	Arranges the given list's elements into a random order
`Collections.sort(list)`	Arranges the given list's elements into sorted order

Custom Ordering with Comparators

Sometimes you want to search or sort a collection of objects in an ordering different that that of its Comparable implementation. For example, consider the following code that sorts an array of Strings and prints the result:

```
String[] strings = {"Foxtrot", "alpha", "echo", "golf",
        "bravo", "hotel", "Charlie", "DELTA"};
Arrays.sort(strings);
System.out.println(Arrays.toString(strings));
```

The following output is produced, which may not be what you expected.

```
[Charlie, DELTA, Foxtrot, alpha, bravo, echo, golf, hotel]
```

Notice that the elements are in case-sensitive alphabetical ordering, with all uppercase strings coming before all lowercase ones. The problem is that the compareTo method of String objects uses case-sensitive ordering. It's conceivable that we might want to sort in many orders, such as by length, by a case-insensitive ordering, or even in reverse order.

It's possible to define your own orderings for objects with special objects named *comparators* that implement comparisons between pairs of objects. The interface Comparator in the java.util package has a method named compare that describes such a comparison. The Arrays.sort, Collections.sort, and Arrays.binarySearch methods all have variations that accept a Comparator as an additional parameter. This comparator will be used to determine the ordering between the elements of your array or collection.

You can implement your own comparators or use some that come with Java's class libraries. For example, if you'd like to sort Strings in case-insensitive alphabetical order, there's a static constant Comparator object in the String class named CASE_INSENSITIVE_ORDER that you can use:

```
String[] strings = {"Foxtrot", "alpha", "echo", "golf",
        "bravo", "hotel", "Charlie", "DELTA"};
Arrays.sort(strings, String.CASE_INSENSITIVE_ORDER);
System.out.println(Arrays.toString(strings));
```

The array is output in the following order:

```
[alpha, bravo, Charlie, DELTA, echo, Foxtrot, golf, hotel]
```

Implementing a Comparator is much like implementing the Comparable interface, except that instead of placing the code inside the class to be compared, you write it as its own class that accepts two parameters for the objects to compare. Comparator is a generic interface that must be told the type of objects you'll be comparing, such as Comparator<String> or Comparator<Point>.

As you recall from the implementation of the Comparable interface, we should return a negative number if the first parameter comes earlier in order than the second; 0 if they are equal; and a positive number if the first parameter comes later than the second. Recall that implementing Comparable in Chapter 10, we saw that a compareTo method examining integer data can often simply subtract one number from the other and return the result. Our compare method is also based on this idea.

```
1  import java.util.*;
2
3  // Compares String objects by length.
4  public class LengthComparator implements Comparator<String> {
5      public int compare(String s1, String s2) {
6          return s1.length() - s2.length();
7      }
8  }
```

Now that we've written a length comparator, we can pass one when sorting an array or list of String objects.

```
Arrays.sort(strings, new LengthComparator());
System.out.println(Arrays.toString(strings));
```

Here's the output when used on the same String array from earlier in this section. Notice that the strings appear in order of increasing length.

```
[echo, golf, alpha, bravo, hotel, DELTA, Foxtrot, Charlie]
```

Sometimes you want to search or sort a collection of objects that don't implement the Comparable interface. For example, the Point class doesn't implement Comparable, but you might wish to sort an array of Point objects by x-coordinate, breaking ties by y-coordinate. An example Comparator that compares Point objects in this way is the following:

```
 1  import java.awt.*;
 2  import java.util.*;
 3
 4  // Compares Point objects by x coordinate and then by y coordinate.
 5  public class PointComparator implements Comparator<Point> {
 6      public int compare(Point p1, Point p2) {
 7          int dx = p1.x - p2.x;
 8          int dy = p1.y - p2.y;
 9
10          if (dx < 0 || (dx == 0 && dy < 0)) {
11              return -1;
12          } else if (dx > 0 || (dx == 0 && dy > 0)) {
13              return 1;
14          } else {
15              return 0;
16          }
17      }
18  }
```

The following shorter version using the compareTo subtraction trick also works.

```
public int compare(Point p1, Point p2) {
    int dx = p1.x - p2.x;
    int dy = p1.y - p2.y;

    if (dx == 0) {
        return dy;
    } else {
        return dx;
    }
}
```

For example, the following code uses our PointComparator to sort an array of four Point objects.

```
Point[] points = {
    new Point(4, -2),
    new Point(-1, 15),
    new Point(3, 7),
    new Point(0, 9)
};
Arrays.sort(points, new PointComparator());
```

After this code, the points appear in the following order:

```
(-1, 15), (0, 9), (3, 7), (4, -2)
```

The following table summarizes several useful places Comparators appear in the Java class libraries.

Useful Comparator Functionality in Java's Class Libraries

Comparator/Method	Description
`Arrays.binarySearch(array, value, comparator)`	Returns the index of the given value in the given array, assuming that the array is currently sorted in the comparator's order
`Arrays.sort(array, comparator)`	Sorts the given array in the ordering of the given comparator
`Collections.binarySearch(list, value, comparator)`	Returns the index of the given value in the given list, assuming that the list is currently sorted in the ordering of the given comparator
`Collections.max(collection, comparator)`	Returns the largest value of the collection according to the ordering of the given comparator
`Collections.min(collection, comparator)`	Returns the smallest value of the collection according to the ordering of the given comparator
`Collections.sort(list, comparator)`	Sorts the given list in the ordering of the given comparator

Comparators can also be used with various collections that use element ordering, such as TreeSet or PriorityQueue.

13.2 Program Efficiency

As we progress in this textbook, the programs we're writing are getting more complex. We're also seeing that there are many ways to solve the same problem. How do we compare different solutions to the same problem, to see which is better?

Also, we say that an algorithm that solves a problem quickly and cleverly is *efficient*. But how fast is fast enough? An algorithm that looks up a word in dictionary and takes 5 minutes to run is probably too slow. But an algorithm that renders a complex 3-dimensional animation in 5 minutes is probably very fast. The acceptable performance for a given program depends on what the program is doing.

So how do we gauge computing performance? One way is to write the program, run it, and measure how long it takes to run. This is sometimes called an *empirical analysis* of the algorithm.

For example, which is the faster program to search an array: one that sequentially searches for the desired target element, or one that first sorts the array, then performs a binary search on the sorted array? We could empirically analyze it by writing both programs, running them and timing them to determine which performs better.

But empirically analyzing an algorithm isn't a very reliable measure, because on a different computer with different processor speed and memory, the program may not run in the same amount of time. Also, to empirically test an algorithm we must write it and time it, which can be a chore.

A more neutral way to measure a program's performance is to examine its code or pseudocode and count roughly how many statements are executed. This is a light version of what's called *algorithm analysis*, which consists of techniques to mathematically predict and compute the performance of various computing algorithms.

It's hard to know exactly how to count how many statements a piece of code executes, because not all statements require the same amount of time to execute. For example, multiplication is slower than addition for a CPU to compute, and a method call is generally slower than evaluating the boolean test of an if/else statement. But for purposes of simplification, let's consider the following actions as requiring exactly 1 unit of time to execute:

- variable declarations and assignments
- evaluating mathematical and logical expressions
- accessing or modifying an individual element of an array
- some method calls

The question arises of how to gauge the runtime of multiple statements placed together. Statements placed in sequential order cause their runtimes to be added together.

```
statement1  \
statement2   > 3
statement3  /
```

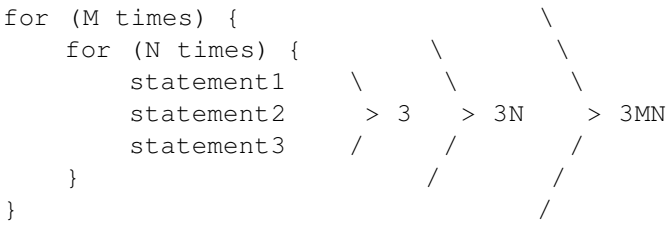
A loop that repeats N times executes roughly N times the number of statements in the body.

```
for (N times) {    \
    statement1      \
    statement2       > 3N
    statement3      /
}                  /
```

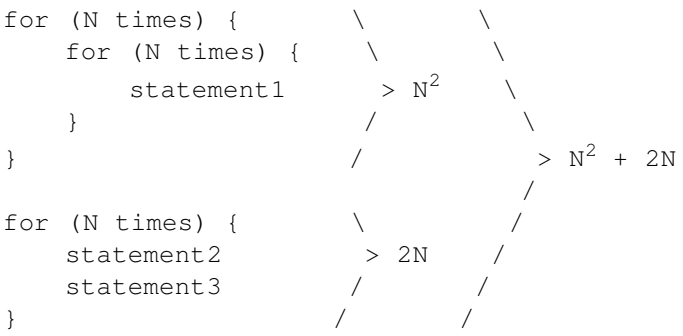
A piece of code with loops placed sequentially with other statements, including other loops, sees the runtimes of those statements added together.

```
statement1                \
for (N times) {    \       \
    statement2      > N     \
}                  /         \
                              > M + N + 3
for (M times) {    \         /
    statement3      > M     /
}                  /       /
statement4                /
statement5               /
```

When two loops are nested, the runtime of the inner loop is multiplied by the number of repetitions of the outer loop.



Normally, the loops in our long-running algorithms are processing some sort of data. Many algorithms run very quickly if the input data size is small, so we generally only worry about the performance for large data sizes. Consider the following loops that examine an array of N elements:



When analyzing code like this, we often think about which line is the most frequently executed in the code. For large values of N, the first N^2 loop executes its statement far more times than the second N loop executes its two statements. For example, if N is 1000, statement1 executes (1000 * 1000) or 1,000,000 times, while statement2 and statement3 each execute only 1000 times. In other words, when two blocks of code appear in sequential order, the block raised to the higher power of N dominates the overall runtime. Statement1 is the most frequently executed in the preceding code.

In fact, when performing algorithm analysis, we often ignore all but the most frequently executed part of the code, because the other parts will generally be outweighed by its runtime. It would be common to refer to the preceding code as an N^2 algorithm rather than an $N^2 + 2N$ algorithm, because the 2N doesn't matter very much if N is large enough. We'll revisit this idea later in the chapter.

A key concept to take away from algorithm analysis is how expensive it is to perform nested loops over large input data. Algorithms that make many passes (especially nested loops) over a very large array or collection will see poor performance, so it's important to come up with efficient algorithms that don't examine the data needlessly. We'll now see these ideas in action as we observe the runtimes of some actual algorithms to solve a programming problem on large data sets.

Algorithm Runtimes

Consider the task of computing the range of numbers in an array. The range is one more than the largest difference between any pair of elements in the array. An initial solution might use nested loops to examine every pair of elements in the array, computing their difference, and remembering the largest difference found.

```
max = 0.
for each index i,
    for each index j,
        if elements [i] and [j] differ by more than max,
            update max.
```

The following code implements the range method as described:

```
// Returns the range of numbers in the given array.
public static int range(int[] numbers) {
    int maxDiff = 0;
    for (int i = 1; i < numbers.length; i++) {
        for (int j = 1; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]) + 1;
            maxDiff = Math.max(maxDiff, diff);
        }
    }
    return maxDiff;
}
```

Since the code has two for loops that each process the entire array, we can hypothesize that the algorithm executes roughly n^2 statements or some multiple thereof.

We can measure the speed of this range algorithm in milliseconds by calling range on various arrays and measuring the time elapsed. The timing is done by acquiring the current time (found using a method named `System.currentTimeMillis`) before and after calling range on a large array, and subtracting the end time from the start time.

```
     n      Range    Runtime (ms)
  1000        219          15
  2000        423          47
  4000        396         203
  8000        194         781
 16000        381        3110
 32000        891       12563
 64000      41549       49937
```

Notice that as the input size N doubles, the runtime of the range method approximately quadruples. This is consistent with our hypothesis. If the algorithm takes n^2 statements to run and we increase the input size to 2n, the new runtime is roughly $(2n)^2$ or $4n^2$, which is 4 times as large as the original runtime.

Our code isn't very efficient for a large array. It needed over 12 seconds to examine 32000 integers. In real-world data processing situations, we would expect to see far larger input data than this. The runtime isn't acceptable for general use.

Looking at a piece of code and trying to speed it up can be deceptively difficult. It's tempting to want to tweak the code by looking at each line and trying to reduce the amount of computation it performs. For example, you may have noticed that our range method actually examines every pair of elements in the array twice. For unique integers i and j, we examine the pair (numbers[i],

numbers[j]) as well as the pair (numbers[j], numbers[i]). We can perform a tweak of our range method code by starting each inner j loop ahead of i, so that we won't examine any pair (i, j) with i >= j.

```
// Returns the largest of all integers in the given array.
public static int range2(int[] numbers) {
    int maxDiff = 0;
    for (int i = 1; i < numbers.length; i++) {
        for (int j = i + 1; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]) + 1;
            maxDiff = Math.max(maxDiff, diff);
        }
    }
    return maxDiff;
}
```

Since about half of the possible pairs of i/j values are eliminated by this tweak, we'd hope that the code would run about twice as fast. The following timing test shows its actual runtime:

```
     n      Range    Runtime (ms)
  1000        598          16
  2000        122          16
  4000        540         110
  8000        548         406
 16000       5910        1578
 32000       1582        6265
 64000      24051       25031
```

As we estimated, the second version is about twice as fast as the first. In fact, there are other minor tweaks we could play with, such as replacing the Math.max call with a simple if test (which speeds up the algorithm by around 10% more). But there's a more important point to be made. When the input size doubles, the runtime of either algorithm roughly quadruples. The effect of this is that regardless of which version we use, if the input array is very large, the method will be too slow!

Rather than trying to further tweak our nested loop solution, let's try to think of a smarter algorithm altogether. The range of values in an array is really just one more than the difference between the array's largest and smallest element. We don't really need to examine all pairs of values, only the pair representing the largest value and smallest value. We can discover both of these values in a single loop over the array. The following new algorithm demonstrates this idea:

```
// Returns the largest of all integers in the given array.
public static int range3(int[] numbers) {
    int max = numbers[0];
    int min = max;
    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] > max) {
            max = numbers[i];
        } else if (numbers[i] < min) {
            min = numbers[i];
        }
    }

    return max - min + 1;
}
```

Since this algorithm only passes over the array once, we'd hope that its runtime is only proportional to the array's length. If the array length doubles, the runtime should double, not quadruple. The following timing test shows its actual runtime:

```
         n      Range    Runtime (ms)
      1000        121              0
      2000        152              0
      4000       1739              0
      8000       1431              0
     16000        548              0
     32000      16534              0
     64000      16458              0
    128000      26128              0
    256000      43257              0
    512000      78586              0
   1024000      62718              0
   2048000     501992             16
   4096000    3228947             31
   8192000     621097             47
  16384000    2309851             94
  32768000    7856008            188
  65536000   39748656            453
 131072000    7021645            797
 262144000   51030232           1578
```

Our runtime approximations were roughly correct. As the size of the array doubles, the runtime of this new range algorithm roughly double as well. The overall runtime of this algorithm is much better; we can examine over a hundred million integers in under one second.

There are a few very important observations to take away from the preceding algorithms and their respective runtimes:

- Tweaking an algorithm's code often isn't as powerful an optimization as finding a better algorithm.
- An algorithm's *rate of growth*, or the amount its runtime increases when the input data size increases, is one of the most important measures of the efficiency of the algorithm.

The idea of growth rate is similar to looking only at the most frequently executed line of code in an algorithm analysis and is an important idea when we wish to categorize algorithms by efficiency.

Complexity Classes

We categorize rates of growth based on their proportion to the input data size N. We call these categories *complexity classes*.

Complexity Class

A category of algorithm efficiency based upon its relationship to the input data size.

The complexity class of a piece of code is determined by looking at the most frequently executed line of code, determining the number of times it is executed, and extracting the highest power of N. For example, if the most frequent line executes ($2N^3 + 4N$) times, we say that the algorithm is in the "order N^3" complexity class, or $O(N^3)$ for short. The shorthand notation with the capital O is called *big-Oh notation* and is used commonly in algorithm analysis.

Here are some of the most common complexity classes, listed in order from slowest to fastest growth (from most to least efficient):

- *Constant-time* or O(1) algorithms are those whose runtime doesn't depend on any input size. One such algorithm is the code to read an amount of pennies from the user and report how many quarters can be made from those pennies.

- *Logarithmic* algorithms, or O(log N) for short, divide a problem space repeatedly in half until it is solved. An example logarithmic-time algorithm is one where the user thinks of a number between 1 and 1,000,000 and the computer tries to guess it, giving the computer a hint each time about whether its guess was too high or too low. On each guess the computer can guess the middle number of its current range and cut the range in half based on whether it was too high or too low.

- *Linear* "order N" algorithms, or O(N) for short, are those whose runtime is directly correlated to N (one whose runtime roughly doubles when N doubles). The last version of the range method in the previous section is a linear algorithm.

- Performing a logarithmic operation over every element of an input dataset of size N leads to an algorithmic runtime of O(N log N), sometimes called *log-linear* algorithms.

- *Quadratic* or $O(N^2)$ algorithms are those whose runtime roughly quadruples when N doubles. The initial versions of the range algorithm were quadratic algorithms.

- *Cubic* or $O(N^3)$ algorithms generally make three levels of nested passes over the input data. The task of counting the number of colinear trios of points in a large Point array would be a cubic algorithm.

- *Exponential* algorithms require roughly 2^N operations to perform. One example is the task of printing the "power set" of a data set, which is the set of all possible subsets of the data. Exponential algorithms are so slow that they should only be executed on small input data sizes.

The following table presents several hypothetical algorithm runtimes as an input size N grows. Notice that even if they all start at the same runtime for a small input size, the ones in higher complexity classes quickly become so slow as to be impractical.

input size (N)	O(1)	O(log N)	O(N)	O(N log N)	$O(N^2)$	$O(N^3)$	$O(2^N)$
100	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms
200	100 ms	115 ms	200 ms	240 ms	400 ms	800 ms	32.7 sec
400	100 ms	130 ms	400 ms	550 ms	1.6 sec	6.4 sec	12.4 days
800	100 ms	145 ms	800 ms	1.2 sec	6.4 sec	51.2 sec	36.5 million years
1600	100 ms	160 ms	1.6 sec	2.7 sec	25.6 sec	6 min 49.6 sec	$42.1 * 10^{24}$ years
3200	100 ms	175 ms	3.2 sec	6 sec	1 min 42.4 sec	54 min 36 sec	$5.6 * 10^{61}$ years

For large data sets it's very important to choose the most efficient algorithms possible, in terms of complexity class. If you know the algorithm you want to run is $O(N^2)$ or worse, you probably shouldn't call it on extremely large data sets. We'll now examine algorithms to search and sort data, keeping this lesson in mind.

13.3 Implementing Searching Algorithms

In this section we'll implement methods that search for an integer in an array of integers and return the index where it was found. If the integer doesn't appear in the array, we'll return a negative number. We'll examine two major searching algorithms, sequential and binary search, and discuss the tradeoffs between them.

Sequential Search

Perhaps the simplest way to search an array is to loop over the elements of the array and check each one to see if it is the target number. As mentioned earlier, this is called a sequential search because it examines every element in sequence.

The code for a sequential search is relatively straightforward using a for loop. We'll return -1 if the for loop completes without finding the target number.

```
// Sequential search algorithm.
// Returns the index at which the given target number first
// appears in the given input array, or -1 if it is not found.
public static int indexOf(int[] numbers, int target) {
    for (int i = 0; i < numbers.length; i++) {
        if (numbers[i] == target) {
            return i;   // found it!
        }
    }

    return -1;   // not found
}
```

Based on the rules stated in the previous section, we predict that the sequential search algorithm is a linear, O(N) algorithm because it contains one loop that traverses at most N elements in an array. (We say "at most" because if the algorithm finds the target element, it stops and immediately returns.) Next we'll time it to verify our prediction.

The following runtime table shows actual results of running the sequential algorithm on randomly generated arrays of integers.

n	Index	Runtime	Index	Runtime
400000	10789	0	-1	0
800000	328075	0	-1	0
1600000	249007	0	-1	16
3200000	1410489	15	-1	16
6400000	2458760	16	-1	31
12800000	5234632	37	-1	47
25600000	7695239	32	-1	93
51200000	5096538	45	-1	203
102400000	8938751	72	-1	422
204800000	29198530	125	-1	875

When searching for an integer that isn't in the array, the algorithm runs somewhat slower because it can't bail out early upon finding the target. This raises the question of whether we should judge the algorithm by its fastest or slowest runtime. Often we care most about the expected behavior when fed a random input, often called an *average case analysis*. But in certain conditions, we also care about the fastest possible outcome, the *best case analysis*, and/or the slowest possible outcome, the *worst case analysis*.

Binary Search

Consider a modified version of the searching problem, where we can assume that the elements of the input array are in sorted order. Does this affect our algorithm? Our existing algorithm will still work correctly, but now we can actually stop searching if we ever get to a number larger than our target without finding the target. For example, if we're searching the array `[1, 4, 5, 7, 7, 9, 10, 12, 56]` for the target number 8, we can stop searching once we see the 9.

It might seem that such a modification to our sequential search algorithm would lead to a significant speedup. But in actuality it doesn't make much difference. The only case in which it speeds up the algorithm noticeably is when searching for a relatively small value that isn't found in the array. In fact, when searching for a large value, one that requires the code to examine most or all of the array elements, the modified algorithm actually performs more slowly than the original because it has to perform a few more boolean tests. Most importantly, the algorithm is still O(N), which runs reasonably quickly but isn't the optimal solution.

Once again an algorithm tweak doesn't make as much difference as finding a smarter algorithm in the first place. If we know that our input array is in sorted order, sequential search isn't the best choice. Would you look up someone's phone number in the phone book by turning to the first page, then the second, and so on until you found the phone number? Of course not. You know that the pages are sorted by name, so you'd flip open the book to somewhere near the middle of the book, then narrow down toward the first letter of the person you were looking for.

The binary search algorithm discussed previously in this chapter takes advantage of the sortedness of the array. Binary search keeps track of the range of the array that is currently of interest. (Initially this range is the whole array.) The algorithm repeatedly examines the center element of the array, and uses this to eliminate half of the range of interest. If the center element is smaller than the target we're searching for, we eliminate the lower half of the range. If the center element is larger than the target, we eliminate the upper half.

We can implement the preceding idea by keeping track of three indexes that will change as our algorithm runs: the middle index, the minimum index to examine, and the maximum index. If we examine the middle element and it's too small, we will eliminate all elements between mid and min from consideration. If the middle element is too large, we will eliminate all elements between min and max from consideration. Consider the following array:

```
int[] numbers = {11, 18, 29, 37, 42, 49, 51, 63, 69, 72, 77, 82, 88, 91, 98};
```

Let's binary search the array for a target value of 77. We'll start at the middle element, which is the one at index (size / 2) or 15 / 2 or 7. The following diagrams show the min, mid, and max at each step of the algorithm:

```
index     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
element [11, 18, 29, 37, 42, 49, 51, 63, 69, 72, 77, 82, 88, 91, 98]
          ^                           ^                           ^
step 1   min                        mid                         max

numbers[mid] is too small; move right.
```

```
index     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
element [11, 18, 29, 37, 42, 49, 51, 63, 69, 72, 77, 82, 88, 91, 98]
                                      ^           ^           ^
step 2                              min         mid         max

numbers[mid] is too large; move left.
```

```
index     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
element [11, 18, 29, 37, 42, 49, 51, 63, 69, 72, 77, 82, 88, 91, 98]
                                      ^   ^   ^
step 3                              min mid max

numbers[mid] is too small; move right.
```

```
index     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
element [11, 18, 29, 37, 42, 49, 51, 63, 69, 72, 77, 82, 88, 91, 98]
                                              ^
step 4                                      min
                                            mid
                                            max

found it!
```

What about when searching for an element that isn't found in the array? Imagine that we'd been searching for the value 78 instead of 77. The steps of the algorithm would have been the same, except on the last fourth pass, we would have seen 77 and not the desired 78. The algorithm would know that it should stop once it's trimmed its range down to a single element and that element is not the target. Another way to describe it is to loop until the min and max have crossed each other.

The following code implements the binary search algorithm as described. Its loop repeats until the target is found or until the min and max have crossed.

```
// Binary search algorithm.
// Returns the index at which the given target number first
// appears in the given input array, or -1 if it is not found.
// pre: array is sorted.
public static int binarySearch(int[] numbers, int target) {
    int min = 0;
    int max = numbers.length - 1;

    while (min <= max) {
        int mid = (max + min) / 2;
        System.out.println(mid + 1);
        if (numbers[mid] == target) {
            return mid;     // found it!
        } else if (numbers[mid] < target) {
            min = mid + 1;  // too small
        } else {   // numbers[mid] > target
            max = mid - 1;  // too large
        }
        mid = (max + min) / 2;
    }

    return -1;   // not found
}
```

The following runtime table shows actual results of running the binary search algorithm on randomly generated arrays of integers.

```
         n      Index    Runtime      Index    Runtime
    400000      15268          0         -1          0
    800000     377855          0         -1          0
   1600000     787480          0         -1          0
   3200000    1267940          0         -1          0
   6400000     764109          0         -1          0
  12800000    2593634          0         -1          0
  25600000    5294376          0         -1          0
  51200000   13978599          0         -1          0
 102400000   44963064          0         -1          0
 204800000    8317859          0         -1          0
```

The results are almost shocking. Binary search is so fast, it rounds down to zero when measured by the computer's clock. While this is an impressive result, it makes it harder for us to empirically examine the runtime. What is the complexity class of the binary search algorithm?

The fact that it finishes so quickly might tempt us to conclude that the algorithm is constant-time or O(1). But it doesn't seem right that a method with a loop in it would take a constant amount of time. In actuality the runtime is related to the input size, because the larger the input, the more times we must chop our min-max range in half to arrive a a single element. We could make the following statement about the number of repetitions as related to the input size N:

$$2^{\text{repetitions}} \sim= N$$

Using some algebra and taking a logarithm base-2 of both sides of the equation:

$$\text{repetitions} \sim= \log_2 N$$

We conclude that the binary search algorithm is in the logarithmic complexity class, or O(log N).

Binary search also doesn't have very much difference between its best and worst cases: in the best case, it lucks out and finds its target value in the middle on the first check. In the worst case, it must perform the full log N comparisons. But since logs are small numbers (the log of 1,000,000 is roughly 20), this is not much of a hit.

The following is an alternative recursive version of the binary search, using recursion concepts as presented in Chapter 12.

```
// Recursive binary search algorithm.
// Returns the index at which the given target number first
// appears in the given input array, or -1 if it is not found.
// pre: array is sorted.
public static int binarySearchR(int[] numbers, int target) {
    return binarySearchR(numbers, target, 0, numbers.length - 1);
}

public static int binarySearchR(int[] numbers, int target, int min, int max) {
    // base case
    if (min > max) {
        return -1;   // not found
    } else {
        // recursive case
        int mid = (max + min) / 2;
        if (numbers[mid] == target) {
            return mid;
        } else if (numbers[mid] < target) {
            return binarySearchR(numbers, target, mid + 1, max);
        } else {
            return binarySearchR(numbers, target, min, mid - 1);
        }
    }
}
```

Some folks don't like recursive versions of methods like binary search because there is a non-recursive solution that's fairly easy to write, and because they have heard that recursion has poor runtime performance. While it's true that recursion can be a bit slow because of the extra method calls it generates, that doesn't pose a problem here. The runtime of the recursive version is certainly

still O(log N) because it's essentially performing the same computation; it's still cutting the input in half at each step. In fact, the recursive version is still fast enough that the computer still can't time it accurately; it produces a runtime of 0 ms even on arrays of tens of millions of integers.

In general, analyzing the runtime of recursive algorithms is tricky and requires understanding of a technique called recurrence relations. That's a topic for a later course that won't be covered in this textbook.

Did you Know: Some Quirks of Binary Search

There are a few interesting things we didn't mention yet about binary search, particularly Sun's implementation found in Arrays.binarySearch and Collections.binarySearch. Take a look at this text from the Javadoc documentation of Sun's binarySearch method:

> "The array **must** be sorted (as by the sort method, above) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found."

Binary search depends on the array being sorted. If it isn't sorted, Sun says the results are undefined. What does that mean? Why doesn't Sun just sort the array for you if it's unsorted?

There are two problems with that idea, both essentially related to runtime performance. For one, sorting takes much longer (O(N log N) time) than binary searching (O(log N) time). Second, to even discover that one needs to search requires looking at each element to ensure that they're in order, which takes O(N) time. Essentially, the cost would be too great to examine the array and sort it if necessary.

Even if the cost weren't so large to sort the array, the client of the binarySearch method probably doesn't wish to have their array messed with by the binarySearch method. Searching is supposed to be a read-only operation, not one that rearranges the array.

Let's look at the other part of the previous quote. If the array contains multiple elements with the specified value, there is no guarantee which one will be found. It's something we didn't mention earlier when discussing the algorithm, but in the case of duplicates, binary search isn't guaranteed to find the first occurrence of the element, because the moment it zeroes in on an occurrence, it stops.

Here's another interesting blurb from the binarySearch documentation:

> "Returns: index of the search key, if it is contained in the list; otherwise, (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or list.size(), if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found."

Rather than returning -1 on an unsuccessful search, Sun's binarySearch returns a negative number related to the spot where the element would have been if it had been in the array. For example, calling binarySearch for the value 45 on the following array would return -6 because 5 is the index where 45 would have appeared:

```
[11, 18, 29, 37, 42, 49, 51, 63, 69, 72, 77, 82, 88, 91, 98]
```

The reason Sun returns this value instead of -1 is because if you're maintaining a sorted array or list and want to add 45 to it in the proper index to retain sorted order, you can call binarySearch, get the negative result, negate and subtract 1 from it, and voila, you've got the index at which you should insert 45. This is much faster than linearly searching for the place to add the value, or adding the value at the end and re-sorting the array.

We could modify our own binary search code to match Sun's behavior by changing the last line of the method's body to the following:

```
return –min – 1;   // not found
```

Searching Objects

Searching for a particular object in an array of objects requires a few modifications to our searching code from the previous sections. Let's look at a sequential object search first, because it will work with any type of objects. The Most important modification to make to the code is that it must use the equals method. The == operator doesn't correctly compare objects, so the call on equals tests an array element for equality with the target.

```
// Sequential search algorithm.
// Returns the index at which the given target number first
// appears in the given input array, or –1 if it is not found.
public static int indexOf(Object[] objects, Object target) {
    for (int i = 0; i < objects.length; i++) {
        if (objects[i].equals(target)) {
            return i;   // found it!
        }
    }

    return –1;   // not found
}
```

If we'd like to do a binary search on objects, the array or collection must consist of objects that have a natural ordering. In other words, the objects must implement the Comparable interface. One common example would be an array of Strings. Since we can't use relational operators like < and >= on objects, we must call the compareTo method on pairs of String objects and examine its return value. The following code implements the binary search on an array of Strings:

```
// Binary search algorithm that works with String objects.
// Returns the index at which the given target String
// appears in the given input array, or -1 if it is not found.
// pre: array is sorted.
public static int binarySearch(String[] strings, String target) {
    int min = 0;
    int max = strings.length - 1;

    while (min <= max) {
        int mid = (max + min) / 2;
        int compare = strings[mid].compareTo(target);
        if (compare == 0) {
            return mid;     // found it!
        } else if (compare < 0) {
            min = mid + 1;  // too small
        } else {   // compare > 0
            max = mid - 1;  // too large
        }
    }

    return -1;   // not found
}
```

13.4 Implementing Sorting Algorithms

While there are literally hundreds of algorithms to sort data, we'll cover two in detail in this chapter. The first is one of the most intuitive, though poorly-performing, algorithms. The second is one of the fastest sorting algorithms used in practice today.

The code we'll write in this section will only sort arrays of integer values (ints), but it could be adapted to sort Comparable objects such as Strings. The ideas behind such a conversion are the same as covered in the preceding section on searching objects, such as using the compareTo and equals methods to compare objects rather than relational operators like < and >=.

Selection Sort

The *selection sort* algorithm is a well-known sorting technique that makes many passes over an input array to put its elements into sorted order. Each time through a loop, the smallest value is selected and put in the proper place near the front of the array. Given this array:

```
int[] nums = {12, 123, 1, 28, 93, 16};
```

```
   [0]     [1]     [2]     [3]     [4]     [5]
+-------+-------+-------+-------+-------+-------+
|  12   |  123  |   1   |  28   |  93   |  16   |
+-------+-------+-------+-------+-------+-------+
```

How do you put this array into order from smallest to largest? One step, would be to scan the array and find the smallest number. In this list, the smallest is nums[2] which equals 1. You would be closer to your goal if you were to swap nums[0] and nums[2]:

```
   [0]     [1]     [2]     [3]     [4]     [5]
+-------+-------+-------+-------+-------+-------+
|   1   |  123  |  12   |  28   |  93   |  16   |
+-------+-------+-------+-------+-------+-------+
```

The nums[0] now has the value it should and you only have to order the elements [1] through [5]. You can repeat this algorithm, scanning the remaining five elements for the smallest (nums[2], which equals 12) and swapping it with nums[1]:

```
   [0]     [1]     [2]     [3]     [4]     [5]
+-------+-------+-------+-------+-------+-------+
|   1   |  12   |  123  |  28   |  93   |  16   |
+-------+-------+-------+-------+-------+-------+
```

Now nums[0] and nums[1] have the correct values. You can continue for nums[2], scanning the remaining four elements for the smallest (nums[5] which equals 16) and swapping it with nums[2]:

```
   [0]     [1]     [2]     [3]     [4]     [5]
+-------+-------+-------+-------+-------+-------+
|   1   |  12   |  16   |  28   |  93   |  123  |
+-------+-------+-------+-------+-------+-------+
```

This gives you the correct values for nums[0], nums[1], and nums[2]. You can continue this process until all elements have the proper values. Here is an attempt at a pseudocode description of the algorithm:

```
scan elements [0] through [5] for the smallest value.
swap element [0] with the smallest element found in the scan.

scan elements [1] through [5] for the smallest value.
swap element [1] with the smallest element found in the scan.

scan elements [2] through [5] for the smallest value.
swap element [2] with the smallest element found in the scan.

scan elements [3] through [5] for the smallest value.
swap element [3] with the smallest element found in the scan.

scan elements [4] through [5] for the smallest value.
swap element [4] with the smallest element found in the scan.

scan elements [5] through [5] for the smallest value.
swap element [5] with the smallest element found in the scan.
```

The algorithm involves a scan followed by a swap. You repeat the scan/swap five times. You don't need to perform a sixth scan/swap since the sixth element will automatically have the correct value if the first five have the correct values. This pseudocode is clumsy because it doesn't take into account the obvious looping that is going on. Here is a better approximation:

```
for i from 0 to 4:
    scan elements [i] through [5] for the smallest value.
    swap element [i] with the smallest element found in the scan.
```

You can perform the scan with the following:

```
smallest = (lowest array index of interest).
for (all other index values of interest):
    if (value at [index]) < (value at [smallest]),
        smallest = index.
```

You can incorporate this into your larger pseudocode:

```
for i from 0 to 4:
    smallest = i.
    for index going (i + 1) to 5 do
        if element [index] < element [smallest],
            smallest = index.
    swap element [i] with element [smallest].
```

This pseudocode is almost directly translatable into Java except for the swap process. For this, you should use the following method.

```
// Swaps a[i] with a[j].
public static void swap(int[] a, int i, int j) {
    if (i != j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

Now you can write the method properly:

```
// Places the elements of the given array into sorted order
// using the selection sort algorithm.
// post: array is in sorted (nondecreasing) order
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // find index of smallest element
        int smallest = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[smallest]) {
                smallest = j;
            }
        }

        swap(a, i, smallest);  // swap smallest to front
    }
}
```

Since selection sort makes N passes over an array of N elements, its performance is $O(N^2)$. Technically it examines roughly N + (N-1) + (N-2) + ... + 3 + 2 + 1 elements. There's a mathematical identity that the sum of all integers from 1 to any maximum value N equals (N)(N + 1) / 2, which is just over $1/2\ N^2$. The following empirical runtime chart confirms this analysis, because the runtime quadruples every time the input size N is doubled, which is characteristic of an N^2 algorithm:

```
         n    Runtime (ms)
      1000           0
      2000          16
      4000          47
      8000         234
     16000         657
     32000        2562
     64000       10265
    128000       41141
    256000      164985
```

Merge Sort

There are other algorithms similar to selection sort, that make many passes over the array and swap various elements on each pass. Fundamentally algorithms that swap elements into order in this way are limited to $O(N^2)$ average runtime. We need to examine a better algorithm that breaks this barrier.

The *merge sort* algorithm derives from the observation that if you have two sorted subarrays, you can easily merge them into a single sorted array. For example, if you have the following array:

```
int[] nums = {1, 3, 6, 7, 2, 4, 5, 8};
```

You can think of it as being two halves, each of which happens to be sorted.

```
{1, 3, 6, 7, 2, 4, 5, 8}
 |        |  |        |
 +--------+  +--------+
  subarray    subarray
```

The basic idea of the merge sort algorithm is:

```
mergeSort(array):
    if array is larger than 2 elements,
        split array into two halves.
        sort left half.
        sort right half.
        merge halves.
```

Let's look at splitting the array and merging halves first, then lastly we'll talk about the sorting.

Splitting one array into its two halves is relatively straightforward. We'll set a midpoint at 1/2 of the length of the array, and consider everything before this midpoint to be part of the "left" half and everything after to be in the "right" half. Here are two quick methods for extracting the halves out of an array:

```
// Returns the first half of the given array.
public static int[] leftHalf(int[] array) {
    int size1 = array.length / 2;
    int[] left = new int[size1];
    for (int i = 0; i < size1; i++) {
        left[i] = array[i];
    }
    return left;
}

// Returns the second half of the given array.
public static int[] rightHalf(int[] array) {
    int size1 = array.length / 2;
    int size2 = array.length - size1;
    int[] right = new int[size2];
    for (int i = 0; i < size2; i++) {
        right[i] = array[i + size1];
    }
    return right;
}
```

Now let's pretend that the halves are sorted (we'll come back to how to sort them later) and look at how we'd merge two sorted subarrays. Imagine that you have two stacks of exam papers, each sorted by name, and you need to combine them into a single stack sorted by name. The simplest algorithm is to place both stacks in front of you and look at the top paper of each stack. You grab the paper that comes first in alphabetical order and put it face-down into your merged pile. Then you repeat, looking at both input stacks again, until one stack is empty. Once one is empty, you just grab the entire remaining stack and place it at the end of your merged pile.

The idea is similar here, except that instead of physically removing papers (integers) from the piles (subarrays), since that is slow, we'll keep an index into each subarray and increment that index as we process a given element. Here is a pseudocode description of the merging algorithm:

```
left index i1 = 0.
right index i2 = 0.
for (number of elements in entire array):
    if left value at i1 is smaller than right value at i2,
        include value from left array in new array.
        increase i1 by one.
    else,
        include value from right array in new array.
        increase i2 by one.
```

Here is a trace of the eight steps involved in merging the two subarrays into a new sorted array:

```
Subarrays                       Next Include     Merged Array
-------------------------------------------------------------
{1, 3, 6, 7}  {2, 4, 5, 8}      1 from left      {1}
 ^             ^
i1            i2

{1, 3, 6, 7}  {2, 4, 5, 8}      2 from right     {1, 2}
    ^          ^
   i1         i2

{1, 3, 6, 7}  {2, 4, 5, 8}      3 from left      {1, 2, 3}
    ^             ^
   i1            i2

{1, 3, 6, 7}  {2, 4, 5, 8}      4 from right     {1, 2, 3, 4}
       ^          ^
      i1         i2

{1, 3, 6, 7}  {2, 4, 5, 8}      5 from right     {1, 2, 3, 4, 5}
       ^             ^
      i1            i2

{1, 3, 6, 7}  {2, 4, 5, 8}      6 from left      {1, 2, 3, 4, 5, 6}
       ^                ^
      i1               i2

{1, 3, 6, 7}  {2, 4, 5, 8}      7 from left      {1, 2, 3, 4, 5, 6, 7}
          ^              ^
         i1             i2

{1, 3, 6, 7}  {2, 4, 5, 8}      8 from right     {1, 2, 3, 4, 5, 6, 7, 8}
                       ^
                      i2
```

After the seventh step, you have included all of the elements in the left array, so the left index no longer points to anything. This complicates the pseudocode somewhat, because you have to deal with the special case that one of the indexes might not point to anything. The simple test in the pseudocode needs to be expanded:

```
if i2 has passed the end of the right array, or
left element at i1 is smaller than right element at i2,
    take from left.
else,
    take from right.
```

The actual code to implement the merging behavior is the following.

```
// pre : result is empty; left/right are sorted
// post: result contains the result of merging the sorted lists;
//       list1 is empty; list2 is empty
public static void merge(int[] result, int[] left, int[] right) {
    int i1 = 0;   // index into left array
    int i2 = 0;   // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
            (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];    // take from left
            i1++;
        } else {
            result[i] = right[i2];   // take from right
            i2++;
        }
    }
}
```

Now we've written the code to split an array into halves, and the code to merge sorted halves into a sorted whole. But the halves aren't sorted--how do we sort the halves? We could call selection sort on the halves, but there's a much better thing we can do: *merge sort* them. We can recursively call our own merge sort method on the halves, and if it's written correctly, it'll put the halves into sorted order. Our pseudocode would more accurately be written as the following:

```
split array into two halves.
mergeSort left half.
mergeSort right half.
merge halves.
```

If we're making our merge sort algorithm recursive, it needs a base case and a recursive case. The above pseudocode specifies the recursive case. But for the base case, what are the simplest arrays to sort? An array with either no elements or just one element doesn't need to be sorted at all. It takes at least two elements before you can get something in the wrong order, so the simple cases are arrays of length less than 2.

Here is a method that implements the complete merge sort algorithm:

```
// Places the elements of the given array into sorted order
// using the merge sort algorithm.
// post: array is in sorted (nondecreasing) order
public static void mergeSort(int[] array) {
    if (array.length > 1) {
        // split array into two halves
        int[] left = leftHalf(array);
        int[] right = rightHalf(array);

        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(array, left, right);
    }
}
```

Here is a dynamic scope trace of what happens when you call this method on an array of eight elements:

```
int[] array = {3, 7, 6, 1, 5, 4, 8, 2};
mergeSort(array);

           {3,  7,  6,  1,  5,  4,  8,  2}

split      {3,  7,  6,  1} {5,  4,  8,  2}

split      {3,  7} {6,  1} {5,  4} {8,  2}

split      {3} {7} {6} {1} {5} {4} {8} {2}

merge      {3,  7} {1,  6} {4,  5} {2,  8}

merge      {1,  3,  6,  7} {2,  4,  5,  8}

merge      {1,  2,  3,  4,  5,  6,  7,  8}
```

Here's a view of all calls that are made to the mergeSort and merge methods, in order:

```
mergeSort({3, 7, 6, 1, 5, 4, 8, 2})
    mergeSort({3, 7, 6, 1})
        mergeSort({3, 7})
            mergeSort({3})
            mergeSort({7})
            merge({3}, {7})
        mergeSort({6, 1})
            mergeSort({6})
            mergeSort({1})
            merge({6}, {1})
        merge({3, 7}, {1, 6})
    mergeSort({5, 4, 8, 2})
        mergeSort({5, 4})
            mergeSort({5})
            mergeSort({4})
            merge({5}, {4})
        mergeSort({8, 2})
            mergeSort({8})
            mergeSort({2})
            merge({8}, {2})
        merge({4, 5}, {2, 8})
    merge({1, 3, 6, 7}, {2, 4, 5, 8})
```

The following table demonstrates the performance of merge sort.

```
         n    Runtime (ms)
      1000           0
      2000           0
      4000           0
      8000           0
     16000           0
     32000          15
     64000          16
    128000          47
    256000         125
    512000         250
   1024000         532
   2048000        1078
   4096000        2265
   8192000        4781
  16384000        9828
  32768000       20422
  65536000       42406
 131072000       88344
```

First of all, merge sort's performance is much better than that of selection sort. What is its complexity class? It looks almost like an O(N) algorithm, because the runtime slightly more than doubles when we double the array size.

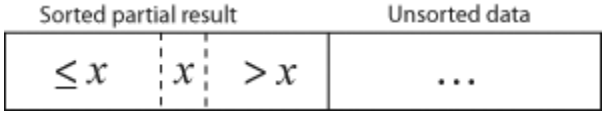
It turns out that merge sort is actually an O(N log N) algorithm. For an array of size n, we have to split it in half roughly $\log_2 N$ times in this algorithm, and at each step we have to do a linear operation of order N (merging the halves after they're sorted). Multiplying these operations produces a O(N log N) overall runtime.

As with other recursive algorithms, a precise analysis of merge sort's performance is complicated and requires mathematical techniques such as recurrence relations, which won't be discussed in this textbook.

Other Sorting Algorithms

Though we won't discuss them in detail here, there are several other important sorting algorithms in common use today.

One sorting algorithm not yet mentioned is *insertion sort*, which is the algorithm we instructors often use when sorting a stack of students' exam papers into alphabetical order. We have a "sorted pile" of papers and consider the rest to be the "unsorted pile." We repeatedly pluck the top paper from the unsorted pile, and insert it into its proper place in the sorted pile. Insertion sort uses this idea to slide elements into place in a growing sorted front portion of an array or list.



Insertion sort's average runtime is $O(N^2)$ like selection sort, but it is somewhat faster overall. Because of this, insertion sort is often used in place of selection sort when a basic n^2 is desired. A nice feature of insertion sort is that it runs very quickly if you run it on an input array that is

already sorted, because no elements need to be moved. One drawback to insertion sort is that it has a poor worst-case runtime, which occurs when the input array is in backwards order. Insertion sort's worst case runtime is worse than that of selection sort.

Another very important and widely used sorting algorithm is named *quicksort*. Invented by influential British computer scientist C.A.R. Hoare, quicksort is an algorithm based on the idea of repeatedly partitioning an array into rough halves. A real-life example of partitioning would be an instructor who has a large pile of exam papers to sort, so she chooses some midpoint of the alphabet and places all the students with "A" through "L" names into one pile, and "M" through "Z" names into another pile. If these piles are still too big to sort, perhaps the instructor further partitions the two piles into two smaller ones: "A" through "F", "G" through "L", "M" through "R", and "S" through "Z". At some point the piles become manageable in size and the teacher and/or assistants can sort each pile, then combine the piles.

Quicksort partitions an array by selecting an element from the array to use as its midpoint (this element is often called the "pivot"). The other elements of the array are split into two groups, one containing all values smaller than the pivot and the other containing all elements greater than the pivot. Once partitioning is completed, the left and right halves are recursively quicksorted. A base case can use a simple sorting algorithm like insertion sort to sort a pile of elements that's smaller than some threshold, such as maybe 100 elements. Eventually the entire array of data falls into sorted order.

Quicksort's average runtime is O(N log N) like merge sort, but it is faster overall. Because of this, quicksort is often used instead of merge sort for top speed. Quicksort also doesn't require a temporary array to be created, which can save a lot of memory for large arrays.

But quicksort does have a few drawbacks that prevent it from being the clear "best" sorting algorithm. One is that if the partitions are very uneven in size (if the pivot is one of the smallest or largest elements and therefore doesn't divide the array very evenly), the sort runs less efficiently, having a worst case performance of $O(N^2)$ in very rare cases. Also, quicksort is a complex algorithm to write, so it can be difficult to produce a working quicksort program without any bugs or mistakes.

The authors chose to cover selection and merge sorts in this textbook because they offer fairly predictable runtime performance and relatively understandable algorithm code.

Chapter Summary

- Searching is the task of attempting to find a particular target value in a collection or array.
- Sorting is the task of arranging the elements of a list or array into a natural ordering.
- Java's class libraries contains several methods for searching and sorting arrays and lists, such as Arrays.binarySearch and Collections.sort.

- A Comparator object describes customized ways to compare objects so that arrays or lists of these objects can be searched and sorted in many orders.
- Empirical analysis is the technique of running a program or algorithm to determine its runtime.
- Algorithm analysis is the technique of examining an algorithm's code or pseudocode to make inferrences about is efficiency.
- Algorithms are grouped into complexity classes, which are often described using big-oh notation, such as O(N) for a linear algorithm.
- Sequential search is a linear O(N) searching algorithm that looks at every element of a list until the target value is found.
- Binary search is a logarithmic O(log N) searching algorithm that operates on a sorted dataset and successively eliminates half of that data until the target element is found.
- Selection sort is a quadratic $O(N^2)$ sorting algorithm that repeatedly finds the smallest unprocessed element of the array and moves it to the frontmost remaining slot of the array.
- Merge sort is a O(N log N) sorting algorithm, often implemented recursively, that successively divides the input dataset into two halves, recursively sorts the halves, and then merges the sorted halves into a sorted whole.

Self-Check Problems

Section 13.1: Searching and Sorting in the Java Class Libraries

1. Describe two ways to search an unsorted array of String objects using the Java class libraries.
2. Should a sequential or binary search be used on an array of Point objects?
3. Under what circumstances can the Arrays.binarySearch and Collections.binarySearch methods be used successfully?

Section 13.2: Program Efficiency

4. Approximate the runtime of the following code fragments, in terms of *n*.

```
int sum = 0;
int j = 1;
while (j <= n) {
    sum++;
    j *= 2;
}
```

```
int sum = 0;
for (int i = 1; i <= n * 2; i = i++) {
    for (int j = 1; j <= n; j++) {
        sum++;
    }
}

for (int j = 1; j < 100; j++) {
    sum++;
    sum++;
}
```

```
int sum = 0;
for (int j = 1; j < n; j++) {
    sum++;
    if (j % 2 == 0) {
        sum++;
    }
}
```

5. Determine the complexity classes of algorithms to perform the following tasks:

   - Finding the average of the numbers in an array of integers
   - Finding the closest distance between any pair of points in a Point array
   - Finding the maximum value in an array of real numbers
   - Counting the median length of the Strings in an array
   - Raising an integer to a power, such as A^B
   - Examining an array of points to see how many trios of points are colinear -- that is, how many groups of three points could be connected by a straight line
   - Counting the number of lines in a file
   - Determining whether a given integer representing year stores a leap year (years divisible by 4 but not by 100 unless divisible by 400)

Section 13.3: Implementing Searching Algorithms

6. What is the runtime complexity class of a sequential search on an unsorted array? What is the runtime complexity class of the modified sequential search on a sorted array?

7. Why does binary search require that the input is sorted?

8. How many elements (at most) does a binary search examine if the array contains 60 elements?

9. What indexes will be examined as the 'middle' element by a binary search for the target value 8 when run on the following input arrays? What value will the binary search algorithm return?

   - `int[] numbers = {1, 3, 6, 7, 8, 10, 15, 20, 30};`
   - `int[] numbers = {1, 2, 3, 4, 5, 7, 8, 9, 10};`
   - `int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};`
   - `int[] numbers = {8, 9, 12, 14, 15, 17, 19, 25, 31};`

10. What indexes will be examined as the 'middle' element by a binary search for the target value 8 when run on the following input array? Notice that the last input array isn't in sorted order. What can you say about the binary search algorithm result?

    - `int[] numbers = {6, 5, 8, 19, 7, 35, 22, 11, 9};`

11. What modifications would have to be made to the selectionSort method to cause it to sort an array of double values rather than one of integer values?

12. How many calls on the mergeSort method are generated by a call to sort a list of length 32?

13. The sorting algorithms in Java's class libraries use quicksort for primitive values, but merge sort for objects. Why is this?

14. Given the following array of int elements:

```
int numbers = {7, 2, 8, 4, 1, 11, 9, 5, 3, 10};
```

    - Show the state of the elements after 5 passes of the outermost loop of selection sort.
    - Show a trace of 2 levels deep of the merge sort algorithm. Show the splitting of the overall array, plus one level deep of the recursive calls.

15. Given the following array of int elements:

```
int numbers = {7, 1, 6, 12, -3, 8, 4, 21, 2, 30, -1, 9};
```

    - Show the state of the elements after 5 passes of the outermost loop of selection sort.
    - Show a trace of 2 levels deep of the merge sort algorithm. Show the splitting of the overall array, plus one level deep of the recursive calls.

Exercises

1. Write code to read a dictionary from a file, then prompt the user for two words, and tell the user how many words in the dictionary fall between those two words. Use binary search in your solution. For example:

```
Type two words: goodbye hello
There are 4418 words between goodbye and hello
```

2. Write a Comparator that compares Point objects by their distance from the origin of (0, 0). Points closer to the origin are considered to come before those further from the origin.

3. Write a Comparator that compares String objects by their number of words. Consider any non-whitespace string of characters as a word. For example, "hello" comes before "I see" comes before "You can do it".

4. Write a modified version of the selection sort algorithm that selects the largest element each time and moves it to the end of the array, rather than selecting the smallest and moving it to the beginning. Will this algorithm be faster than the standard selection sort? What will its complexity class (big-oh) be?

5. Write a modified "dual" version of the selection sort algorithm that selects both the largest and smallest elements on each pass and moves each of them to the appropriate end of the array. Will this algorithm be faster than the standard selection sort? What predictions do you make about its performance relative to the merge sort algorithm? What will its complexity class (big-oh) be?

6. Implement an algorithm to shuffle an array of numbers or objects. The algorithm for shuffling should be the following:

```
for each index i:
    choose a random index j greater than or equal to i.
    swap elements [i] and [j].
```

(The constraint about j being greater than or equal to i is actually quite important, if you want your shuffling algorithm to shuffle fairly. Why?)

Programming Projects

1. Write a program that reads a series of input lines and sorts them in alphabetical order, ignoring the case of words. The program should use the merge sort algorithm so that it quickly sorts even a large file.

2. Write a program that processes a data file of students' course grade data. The data arrives in random order with each line storing information about a student's last name, first name, student ID number, percentage, and letter grade. For example:

```
Smith    Kelly    438975   98.6     A
Johnson  Gus      210498   72.4     C
Reges    Stu      098736   88.2     B
Smith    Marty    346282   84.1     B
Reges    Abe      298575   78.3     C
```

Your program should be able to sort the data by any of the columns. Use Comparators to achieve the sort orderings. Make the data sortable by last name, student ID, and grades in ascending and descending order. For example:

```
Student data, by last name:
Johnson  Gus      210498   72.4     C
Reges    Stu      098736   88.2     B
Reges    Abe      298575   78.3     C
Smith    Kelly    438975   98.6     A
Smith    Marty    346282   84.1     B

Student data, by student ID:
Reges    Stu      098736   88.2     B
Johnson  Gus      210498   72.4     C
Reges    Abe      298575   78.3     C
Smith    Marty    346282   84.1     B
Smith    Kelly    438975   98.6     A
```

3. Write a program that discovers all anagrams of all words, given an input file that stores a large dictionary. An anagram of a word is a rearrangement of its letters into a new legal word. For example, the anagrams of "share" include "shear", "hears", and "hares". Assume that

you have a file available to you which lists many words, one per line. Your program should first read in the dictionary and sort it. It shouldn't sort it in alphabetical order, though. Instead, it should sort according to each word's canonical form. The canonical form of a word is defined to be a word with the same letters as the original, but appearing in sorted order. Thus, the canonical form of "computer" is "cemoprtu" and the canonical form of "program" is "agmoprr". Thus, in sorting the dictionary, the word "program" would be placed before the word "computer", because its canonical form is alphabetically less. Write code to retrieve a word's canonical form and a Comparator that compares words by canonical forms.

Stuart Reges
Marty Stepp

Chapter 14
Graphical User Interfaces

Copyright © 2006 by Stuart Reges and Marty Stepp

- 14.1 Graphical Input and Output with JOptionPane
- 14.2 Graphical Components
  - Working with JFrames
  - Common Components: Buttons, Labels, and Text Fields
  - JTextArea, JScrollPane, and Font
  - Icons
- 14.3 Laying Out Components in a Frame
  - Layout Managers
  - SpringLayout
  - Composite Layout
- 14.4 Events
  - Action Events and ActionListener
  - More Sophisticated ActionEvents
  - A Larger GUI Example with Events: Credit Card GUI
  - Mouse Events
- 14.5 2D Graphics
  - Drawing Onto Panels
  - Simple Animation with Timers
- 14.6 Case Study: Demystifying DrawingPanel

Introduction

In this chapter we will explore the creation of graphical user interfaces (GUIs). While console programs like those we have written in the preceding chapters are still very important, the majority of modern desktop applications have a graphical user interface. In previous chapters, a DrawingPanel was introduced to allow 2D graphics to be drawn on the screen. But writing a GUI is not the same as drawing shapes and lines onto a drawing panel. A real graphical user interface involves the creation of one's own window frames that contain buttons, text input fields, and other onscreen components.

14.1 Graphical Input and Output with JOptionPane

The simplest way to create a graphical window in Java is through the use of a class named JOptionPane. An *option pane* is a simple message box that pops up on the screen and presents a message, or a request for input, to the user. JOptionPane can be thought of as a rough graphical equivalent of System.out.println output and Scanner input.

JOptionPane belongs to the javax.swing package, so you'll need to import this to use it. ("Swing" is the name of Java's GUI libraries.) Note that the package name starts with javax this time, not java (the "x" is because in Java's early days, Swing was an extension to Java's feature set.)

```
import javax.swing.*;  // for GUI components
```

The following program creates a Hello, world! message on the screen.

```
1  // A graphical equivalent of the classic "Hello world" program.
2
3  import javax.swing.*;   // for GUI components
4
5  public class HelloWorld {
6      public static void main(String[] args) {
7          JOptionPane.showMessageDialog(null, "Hello, world!");
8      }
9  }
```

The program produces the following graphical 'output'. (We'll show screenshots for the output of the programs in this chapter.) The window looks slightly different if you're on a different operating system, but the message is the same.

The preceding program uses a static method in the JOptionPane class named showMessageDialog. This method accepts two parameters: a parent window (which we don't have, so we're passing null), and a message string to display.

JOptionPane can be used in three major ways: to display a message (as shown previously), to present a list of choices to the user, and to ask for the user to type input. The three methods that implement these three behaviors are named showMessageDialog, showConfirmDialog, and showInputDialog respectively.

Useful Methods of the JOptionPane class

`showConfirmDialog(parent, message)`
Shows a Yes/No/Cancel message box on screen with the given message on it, returning the choice as an int with one of the following constant values:

- `JOptionPane.YES_OPTION` (user clicked 'Yes')
- `JOptionPane.NO_OPTION` (user clicked 'No')
- `JOptionPane.CANCEL_OPTION` (user clicked 'Cancel')

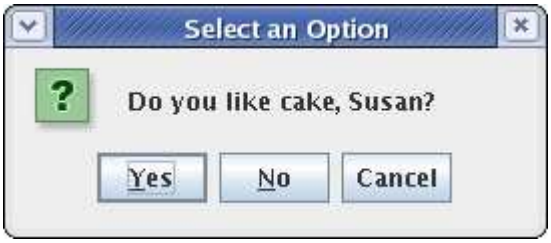
`showInputDialog(parent, message)`
Shows an input box on screen with the given message on it, returning the user's input value as a String.

`showMessageDialog(parent, message)`
Shows the given message string on a message box on screen.

The following program briefly demonstrates all three types of option panes.

```
1  // Shows several JOptionPane windows on the screen.
2
3  import javax.swing.*;   // for GUI components
4
5  public class UseOptionPanes {
6      public static void main(String[] args) {
7          // read the user's name graphically
8          String name = JOptionPane.showInputDialog(null, "What is your name?");
9
10                 // ask the user a yes/no question
11         int choice = JOptionPane.showConfirmDialog(null, "Do you like cake, " + name +
12
13         // show a different response depending on their answer
14         if (choice == JOptionPane.YES_OPTION) {
15             JOptionPane.showMessageDialog(null, "Of course!  Who doesn't?");
16         } else {  // choice == JOptionPane.NO_OPTION or .CANCEL_OPTION
17             JOptionPane.showMessageDialog(null, "We'll have to agree to disagree.");
18         }
19     }
20 }
```

The graphical input and output of this program is a series of windows, which pop up one at a time.



14.2 Graphical Components

JOptionPane is useful, but it is not flexible or powerful enough to create rich graphical user interfaces. To do that, we'll need to learn about the various types of widgets that can be placed on the screen in Java. We call an onscreen window a *frame*. The graphical widgets inside a frame, such as buttons or text input fields, are collectively called *components*.

Frame

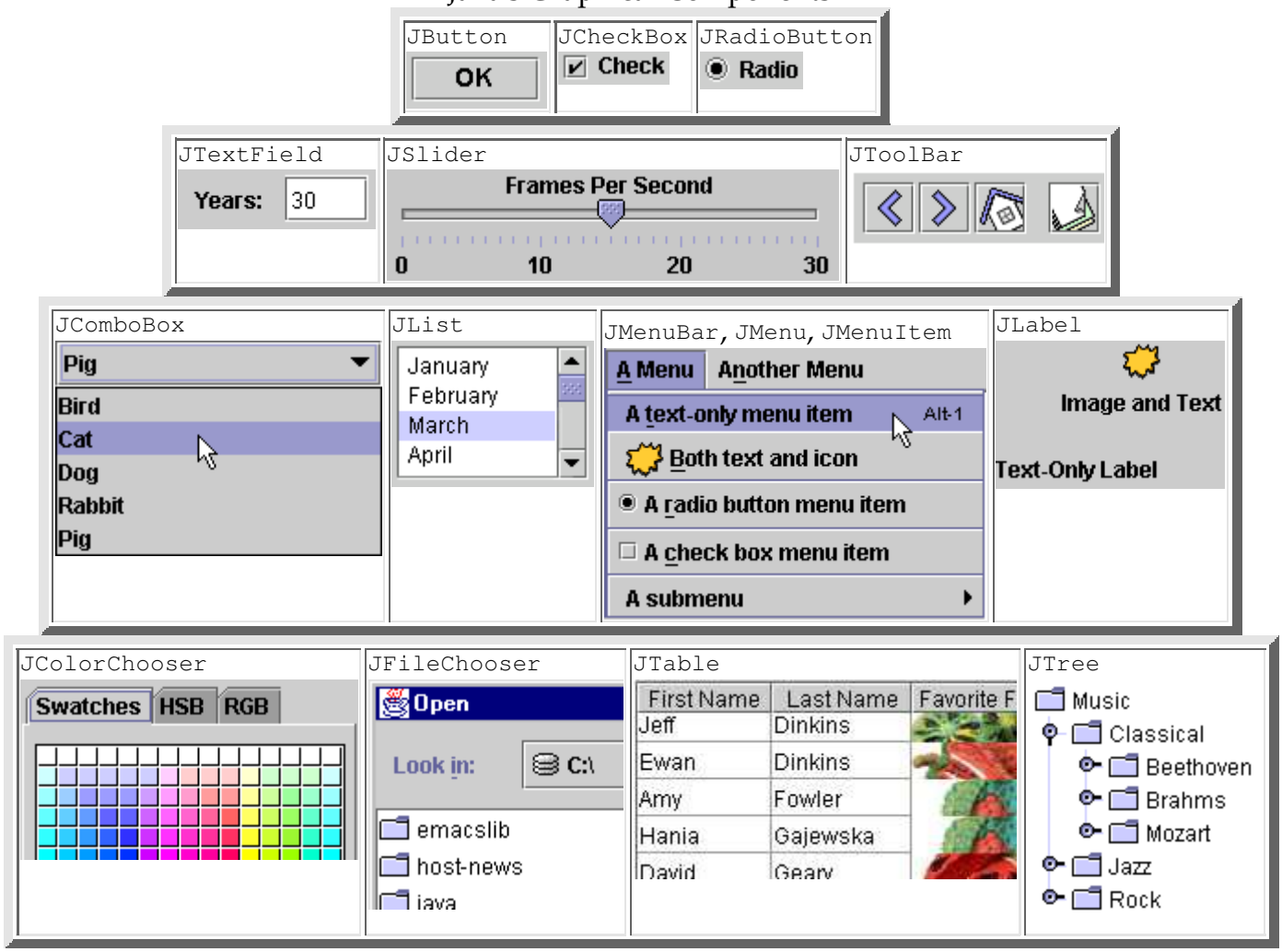
A graphical window on the screen.

Component

A widget, such as a button or text field, that resides inside a graphical window.

Here are some of the more common Java graphical components, listed by class name:

Java's Graphical Components



A more complete pictorial reference of Java's graphical components can be found in Sun's Java Tutorial at:
http://java.sun.com/docs/books/tutorial/uiswing/components/components.html.

Working with JFrames

Any complex graphical program must construct a JFrame object to represent that program's main graphical window. Once a JFrame object has been constructed, we can tell it to display itself on the screen by calling the setVisible method on it, passing the boolean value of true. A simple program that constructs a frame and places it onscreen is the following:

```
1  // Shows an empty window frame on the screen.
2
3  import javax.swing.*;
4
5  public class SimpleFrame {
6      public static void main(String[] args) {
7          JFrame frame = new JFrame();
8          frame.setVisible(true);
9      }
10 }
```

This program produces the following strange graphical output:

There is another problem with the program. When you close the window, it doesn't actually terminate the Java program. When you display JFrames on the screen, Java doesn't exit the program by default when those frames are closed. You can notice that the program hasn't exited because a console window will remain on your screen (if you're using certain Java editors) or because your editor does not show its usual message that the program has terminated. If you do want the program to exit when the window closes, you have to say so explicitly.

To make a more interesting frame, we must learn about the properties that JFrames have. A *property* of a GUI component is a data field or attribute it possesses internally that we may wish to examine or change. A frame's properties control things like the size of the window or the text that appears in the title bar. We can set or examine these properties' values by calling methods on the JFrame.

Here is a list of several JFrame properties:

Useful Properties Specific to JFrames

Property	Type	Description	Methods
default close operation	`int`	What the JFrame should do when it is closed. Possible choices: • `JFrame.DO_NOTHING_ON_CLOSE`: Don't do anything. • `JFrame.HIDE_ON_CLOSE`: Hide the frame. • `JFrame.DISPOSE_ON_CLOSE`: Hide and destroy the frame so that it cannot be shown again. • `JFrame.EXIT_ON_CLOSE`: Exit the program.	`getDefaultCloseOperation`, `setDefaultCloseOperation(int)`
icon image	`Image`	The icon that appears in the title bar and Start menu or Dock	`getIconImage`, `setIconImage(Image)`
layout	`LayoutManager`	an object that controls the positions and sizes of the components inside this frame	`getLayout`, `setLayout(LayoutManager)`
resizable	`boolean`	whether or not the JFrame allows itself to be resized	`isResizable`, `setResizable(boolean)`
title	`String`	text shown on the JFrame's title bar	`getTitle`, `setTitle(String)`

For example, to set the title text of the frame in the last program to "A window frame", you'd write:

```
frame.setTitle("A window frame");
```

It turns out that all graphical components and frames share a common set of properties, because they exist in a common inheritance hierarchy. The following is a partial list of useful properties of frames/components and their respective methods:

Useful Properties of All Components (Including JFrames)

Property	Type	Description	Methods
background	`Color`	background color	`getBackground, setBackground(Color)`
enabled	`boolean`	whether or not the component can be interacted with	`isEnabled, setEnabled(boolean)`
focusable	`boolean`	whether the keyboard may send input to the component	`isFocusable, setFocusable(boolean)`
font	`Font`	font used to write text	`getFont, setFont(Font)`
foreground	`Color`	foreground color	`getForeground, setForeground(Color)`
location	`Point`	(x, y) coordinate of component's top-left corner	`getLocation, setLocation(Point)`
size	`Dimension`	the current width and height of the component	`getSize, setSize(Dimension)`
preferred size	`Dimension`	the 'preferred' width and height of the component; the size it would like to be to make it appear naturally on the screen (used with layout managers, seen later)	`getPreferredSize, setPreferredSize(Dimension)`
visible	`boolean`	whether or not the component can be seen on the screen	`isVisible, setVisible(boolean)`

There are a few new types of objects in the preceding table. The background and foreground properties are Color objects, which we've seen before. The location property is a Point object, which we've also seen. The font property is a Font object; we'll discuss fonts later in this chapter. All of these types are found in the java.awt package, so be sure to import it, just like when drawing graphical programs with DrawingPanel in previous chapters.

```
import java.awt.*;  // for various graphical objects
```

The size property is an object of type Dimension, which simply stores a width and height (and is constructed with two integers representing those values). We'll use this property several times in this chapter. We only need to know a bit about the Dimension type, such as how to construct it.

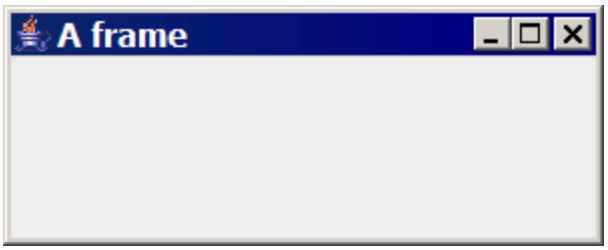
Useful Methods of Dimension objects

`public Dimension(int width, int height)` Constructs a Dimension representing the given size.
`public int getWidth()` Returns the width of this Dimension.
`public int getHeight()` Returns the height of this Dimension.

The following new version of our program creates a JFrame and sets several of the preceding properties. We'll give the frame a color, set its location and size on the screen, and place text into its title bar. Another important property we'll set is the 'default close operation' of the frame, telling it to shut down our Java program when it is closed.

```
1  // Sets several properties of a window frame.
2
3  import java.awt.*;      // for Dimension
4  import javax.swing.*;   // for GUI components
5
6  public class SimpleFrame2 {
7      public static void main(String[] args) {
8          JFrame frame = new JFrame();
9          frame.setForeground(Color.WHITE);
10         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11         frame.setLocation(new Point(10, 50));
12         frame.setSize(new Dimension(300, 120));
13         frame.setTitle("A frame");
14         frame.setVisible(true);
15     }
16 }
```

When the program is run, the following window appears. When the window is closed, the program exits.



Common Components: Buttons, Labels, and Text Fields

Our empty JFrames are not very interesting without anything inside them. Let's look at some graphical components that can be placed in a frame, such as buttons and text fields.

The first component we'll examine is the button, represented by the JButton class. Each JButton object you create represents one button on the screen; if you want three buttons, you must create three JButton objects and place them in your frame.

JButton objects

A button can be constructed either with no parameters (a button with no text), or with a String representing the text on the button. Of course, we can always change the button's text by calling the setText method on it. We can also set other properties of the button such as its background color.

```
JButton button1 = new JButton();
button1.setText("I'm the first button");

JButton button2 = new JButton("The second button");
button2.setBackground(Color.YELLOW);
```

Two other components of interest are the label and the text field. A label is represented by a JLabel object, which can be constructed with an optional parameter specifying the label's text. Text fields are represented by JTextField objects. A JTextField can be constructed by passing it the number of characters wide that should be able to fit in the text field.

```
JLabel label = new JLabel("This is a label");
JTextField field = new JTextField(8);  // 8 letters wide
```

JLabel and JTextField objects

Merely constructing the various component objects does not place them onto the screen. We must add them to our frame so it will display them. A frame acts as a *container*, meaning that it's a region to which you can add graphical components. To add a component to a JFrame, call the frame's `add` method and pass the appropriate component.

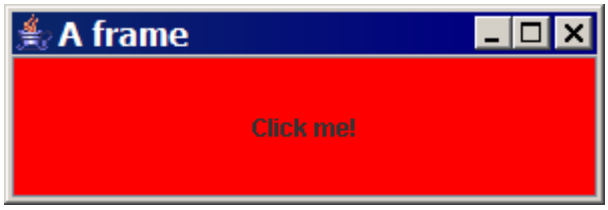
The following program creates a frame and places two buttons inside it:

```
 1  import java.awt.*;
 2  import javax.swing.*;
 3
 4  public class ComponentsExample1 {
 5      public static void main(String[] args) {
 6          JFrame frame = new JFrame();
 7          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 8          frame.setSize(new Dimension(300, 100));
 9          frame.setTitle("A frame");
10
11          JButton button1 = new JButton();
12          button1.setText("I'm a button.");
13          button1.setBackground(Color.BLUE);
14          frame.add(button1);
15
16          JButton button2 = new JButton();
17          button2.setText("Click me!");
18          button2.setBackground(Color.RED);
19          frame.add(button2);
20
21          frame.setVisible(true);
22      }
23  }
```

You'll notice a pattern here. When we create a component, we must do the following things:

- Construct it.
- Set its properties, if necessary.
- Add it to a container on the screen (in our case, the frame).

The program produces the following strange graphical output:



Notice that the first button isn't visible. The second button has been stretched to fill the entire frame! This is because we haven't told the frame how to position and lay out the buttons, so it defaults to giving each component the entire frame's space. The last component added 'wins' and gets all the space.

Each button has a size and location property, but setting these won't fix the problem. The frame has an internal *layout manager* object that it uses to forcibly position all of the components inside it. Even if we set the position and size of the buttons, the frame will set them back to what it prefers.

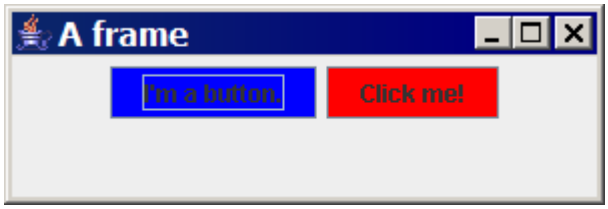
Layout Manager

A Java object that decides the positions, sizes, and resizing behavior of the components within a container on the screen.

The solution to this problem, instead, is to set a new layout manager for the JFrame and instruct it how to position the components. If we want the buttons to flow in a left-to-right order, for example, we can use a FlowLayout. We'd insert the following line in our program before adding the buttons:

```
frame.setLayout(new FlowLayout());
```

After setting the layout, the components now all show up on the screen, though perhaps not exactly in the positions we'd like. The following graphical output is produced:



We'll discuss layout managers in detail in the next section. For now, we will use a FlowLayout only so that we can focus on the components.

Another thing you may notice is that the frame isn't sized quite right to fit its buttons--it has some extra space hanging on the bottom. JFrames have a method named pack that tells them to resize themselves exactly to fit their contents. It's useful to call this method just before showing the frame on the screen, to make sure it's a snug size. Of course, this undoes whatever setSize calls you made previously.

```
frame.pack();
frame.setVisible(true);
```

The new, packed frame has the following onscreen appearance:

The following table summarizes the useful pack method of JFrames:

Useful Methods of JFrame objects

`public void pack()` Resizes the frame to snugly fit the components inside it.

JTextArea, JScrollPane, and Font

While the JTextField component represents a single-line text input field, the JTextArea component represents a multi-line editing box. You can construct a JTextArea by passing the number of rows and columns (the number of lines, and the number of letters in each line).

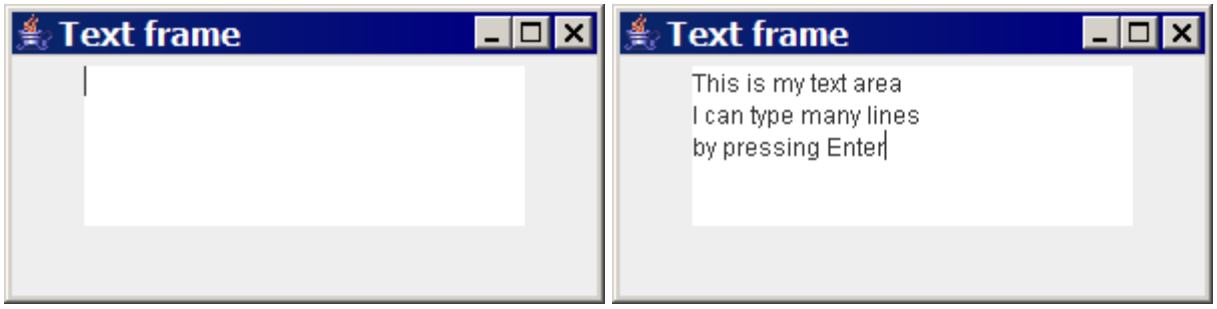
```
// a text area which is 5 lines tall and 20 letters wide
JTextArea area = new JTextArea(5, 20);
```


A JTextArea object

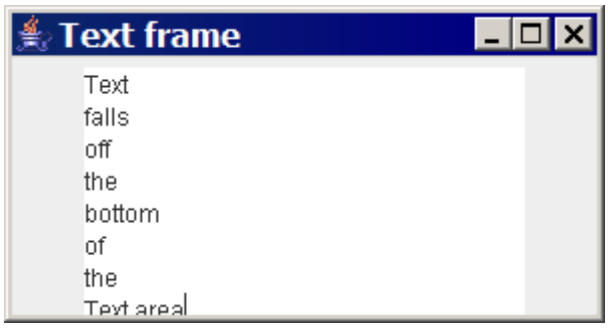
The following program constructs a simple JFrame and adds a JTextArea to it:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class TextFrame {
5      public static void main(String[] args) {
6          JFrame frame = new JFrame();
7          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8          frame.setLayout(new FlowLayout());
9          frame.setSize(new Dimension(300, 150));
10         frame.setTitle("Text frame");
11
12         JTextArea area = new JTextArea(5, 20);
13         area.setFont(new Font("Serif", Font.BOLD, 14));
14         frame.add(new JScrollPane(area));
15
16         frame.setVisible(true);
17     }
18 }
```

The program produces the following graphical output (shown both before and after typing some text onto the text area):



There is one problem with the text area. When the user types too much text to fit in the text area, the text simply vanishes off the bottom of the text area.



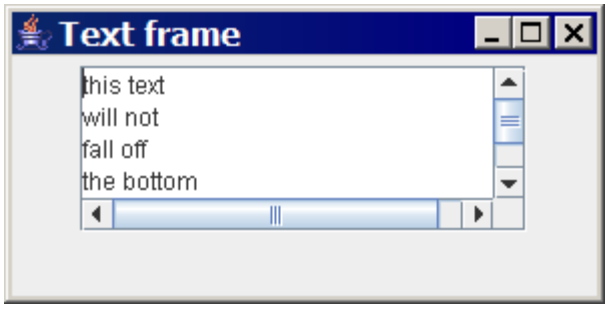
To fix this, we can use a special component called a JScrollPane to attach scroll bars to the text area. A JScrollPane is constructed by passing the text area as a parameter.

Useful methods of JScrollPane objects

`public JScrollPane(Component component)`
Constructs a new scroll pane to add scroll bars to the given component.

We can add the JScrollPane to the frame instead of the JTextArea itself, and now the text area will have scroll bars when its text becomes too big.

```
// use scroll bars on this text area
frame.add(new JScrollPane(area));
```



If you don't like the default font of the text area, remember that every component has a Font property that can be set. A Font object can be constructed by passing its name, style (such as bold or italic), and size in pixels.

Useful methods of Font objects

`public Font(String name, int style, int size)` Constructs a new font of the given style and size.

Font names can refer to a font on your system, such as "Times New Roman", or to special pseudo-font names known to Java such as "Serif", "SansSerif", and "Monospaced". A serif is a fan-out that occurs at the edges of letters in fonts such as Times New Roman, but not in straight-edged or "sans-serif" fonts such as Arial or Verdana. A monospaced font uses evenly sized characters, such as Courier New or your Terminal font. (Most Java editors use a monospaced font to edit your programs so that text columns line up exactly on the screen.)

Font styles can be one or more of the following:

- `Font.BOLD`: thick text.
- `Font.ITALIC`: slanted text.
- `Font.PLAIN`: normal text.

Font styles can be combined by adding them, such as `Font.BOLD + Font.ITALIC` for thick slanted text.

The following line of code tells the text area to use a size 14 serifed font.

```
area.setFont(new Font("Serif", Font.BOLD, 14));
```

Icons

Many Java components, including buttons and labels, have an icon property that can be set to place an image on the component.

Useful Properties of JButton and JLabel objects

Property	Type	Description	Methods
icon	Icon	the icon image that appears on the component	getIcon, setIcon(Icon)

Icon is an interface. The easiest way to get an object that implements Icon is to create an object of type ImageIcon. The ImageIcon constructor accepts a String parameter representing the image file to load. The image is a file on your hard disk, such as a GIF, JPG, or PNG image. Place your image files into the same folder as your .java files.

Useful Methods of ImageIcon objects

`public ImageIcon(String filename)` Constructs an icon from the image in the given file.
`public ImageIcon(Image image)` Constructs an icon from the given image. (See BufferedImage below)

The following code places an icon onto a JButton. The file smiley.jpg has already been saved to the same folder as the code.

```
// create a smiley face icon for this button
JButton button = new JButton("Have a nice day");
button.setIcon(new ImageIcon("smiley.jpg"));
```

When placed into a JFrame, the button has the following appearance. The button is larger to accommodate both its text and its new icon.

Another way to create an icon is to draw one for yourself. You can create a blank image buffer using a BufferedImage object, and draw onto it using a Graphics pen, much like you did with the DrawingPanel in previous chapters. BufferedImage is a class in the java.awt.image package, so if you want to use it, you must import this package.

```
import java.awt.image.*;  // for BufferedImage
```

A BufferedImage plays a key role in the way the DrawingPanel is implemented. Here are the useful constructor and methods from BufferedImage:

Useful Methods of BufferedImage objects

`public BufferedImage(int width, int height, int type)` Constructs an image buffer of the given size and type. Valid types are: • `BufferedImage.TYPE_INT_ARGB`: An image with a transparent background. • `BufferedImage.TYPE_INT_RGB`: An image with a solid black background.
`public Graphics getGraphics()` Returns the Graphics pen for drawing on this image buffer.

To use a BufferedImage, construct one of a particular size and type (we recommend BufferedImage.TYPE_INT_ARGB), then get the Graphics object from it using getGraphics(), then issue the standard drawing commands such as drawRect or fillOval. Once you're done drawing on the BufferedImage, you can set it as the icon for an onscreen component by creating a new

ImageIcon object and passing the BufferedImage to the ImageIcon constructor. You can use this ImageIcon as the icon for an onscreen component by calling setIcon on that component and passing the ImageIcon as the parameter.

```
JButton button = new JButton();
button.setText("My drawing");

// create a shape image icon for this button
BufferedImage image = new BufferedImage(100, 100,
        BufferedImage.TYPE_INT_ARGB);
Graphics g = image.getGraphics();
g.setColor(Color.YELLOW);
g.fillRect(10, 20, 80, 70);
g.setColor(Color.RED);
g.fillOval(40, 50, 25, 25);

ImageIcon icon = new ImageIcon(image);
button.setIcon(icon);
```

A BufferedImage can also be used as the icon for a JFrame, by calling its setIconImage method. This is a case where Java's designers chose confusing names, because the setIconImage method doesn't accept an ImageIcon as its parameter.

```
frame.setIconImage(image);
```

When placed into a JFrame, the button's appearance is the following. Notice how a smaller version of the icon also appears at the top-left of the window, because of the setIconImage call.

14.3 Laying Out Components in a Frame

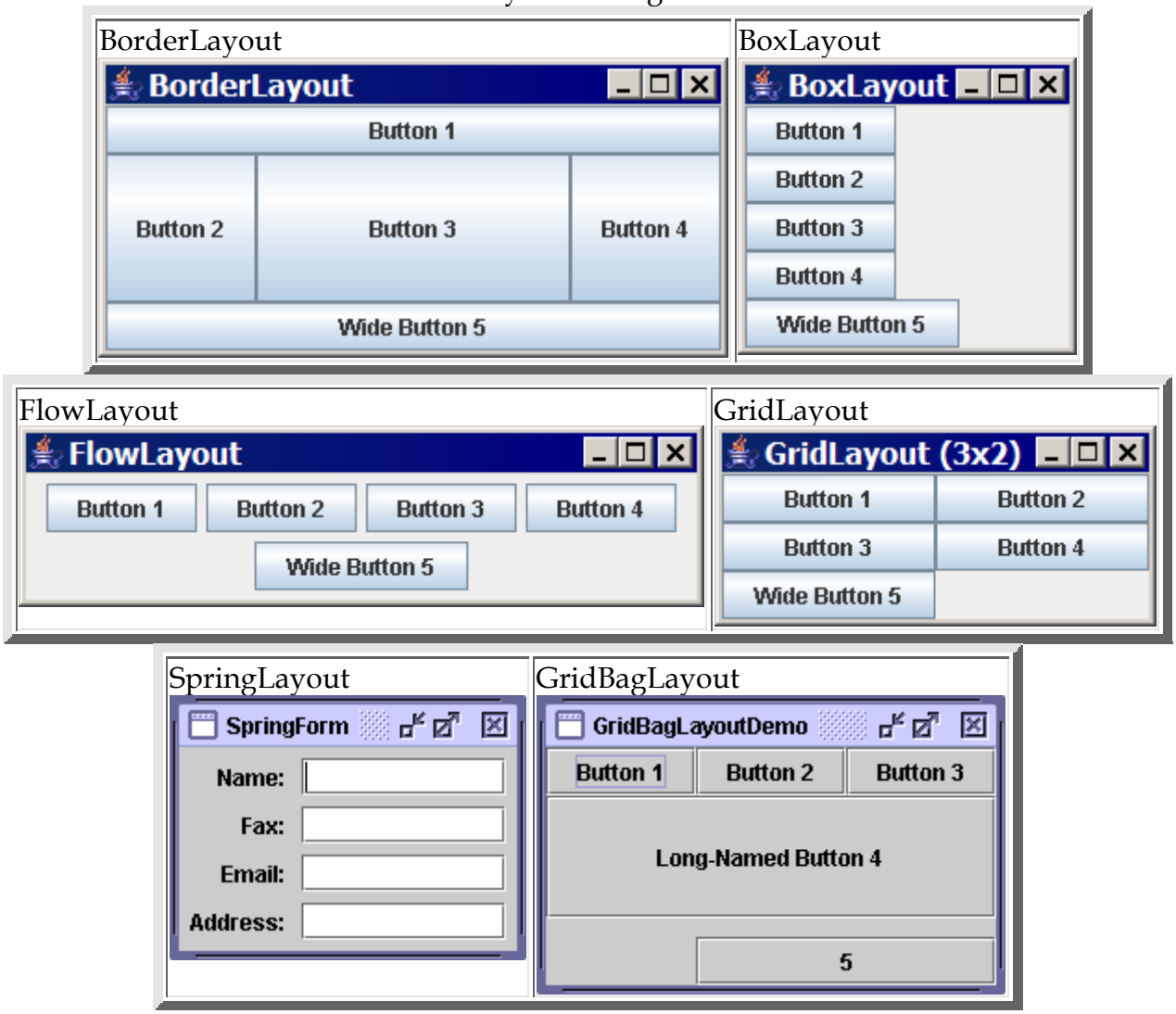
In the past section, we used FlowLayout to position components in our JFrame. In this section, we will explore several different layout manager objects that can be used to position components in a variety of ways.

Layout Managers

Java contains many layout manager classes in its java.awt package, so make sure to import that (along with javax.swing for the JFrame and other component classes).

```
import java.awt.*;     // for LayoutManagers
import javax.swing.*;  // for GUI components
```

Here are some of the most common layout managers:

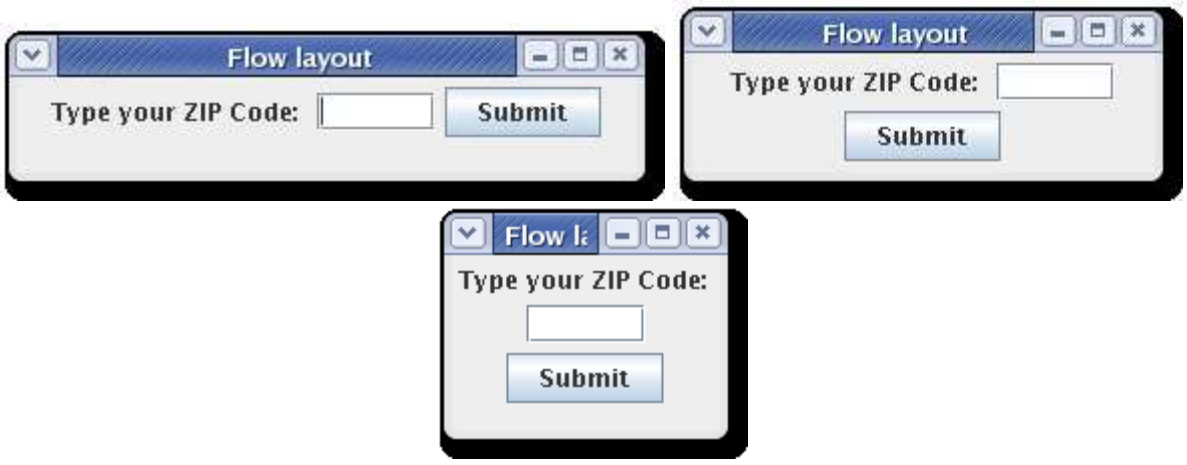


Perhaps the simplest layout manager is FlowLayout. FlowLayout positions components in a left-to-right, top-to-bottom 'flow' that is somewhat like words in a paragraph. If a row of components is too wide to fit in the frame, the row wraps around.

Consider the following code, which adds three components to a frame that uses a FlowLayout:

```
JLabel label = new JLabel("Type your ZIP code: ");
JTextField field = new JTextField(5);
JButton button = new JButton("Submit");

frame.setLayout(new FlowLayout());
frame.add(label);
frame.add(field);
frame.add(button);
```

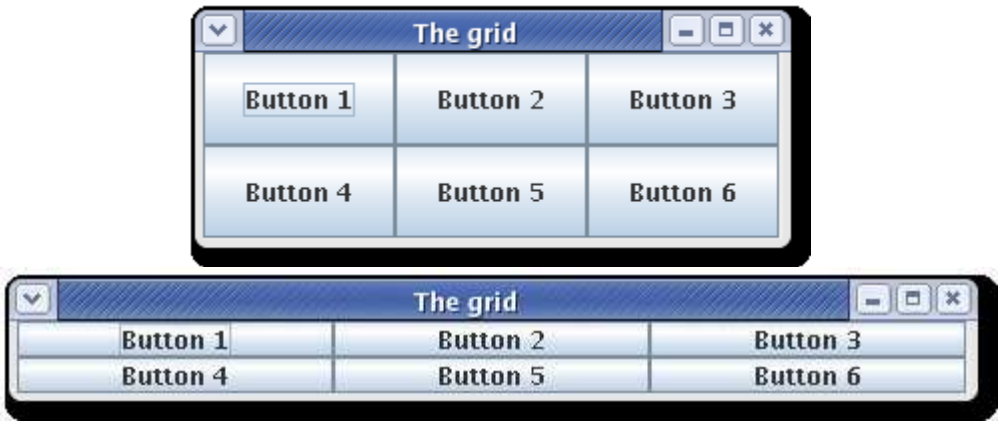
The following graphical output is produced. We've then resized the window to show the wrapping behavior:



Another common layout is named GridLayout. Construct a GridLayout by passing it a number of rows and columns. Now when components are added to the frame, they are laid out in equal-sized squares. The components are placed in the squares in row-major order (left-to-right, top-to-bottom). We'll use only buttons in this example so you can clearly see each component's size and shape:

```
// 2 rows, 3 columns
frame.setLayout(new GridLayout(2, 3));
for (int i = 1; i <= 6; i++) {
    frame.add(new JButton("Button " + i));
}
```

The following graphical output is produced. Notice how GridLayout forces every component to take the same size, even though this results in awkwardly stretched buttons. The window is then resized to further show the stretching behavior:

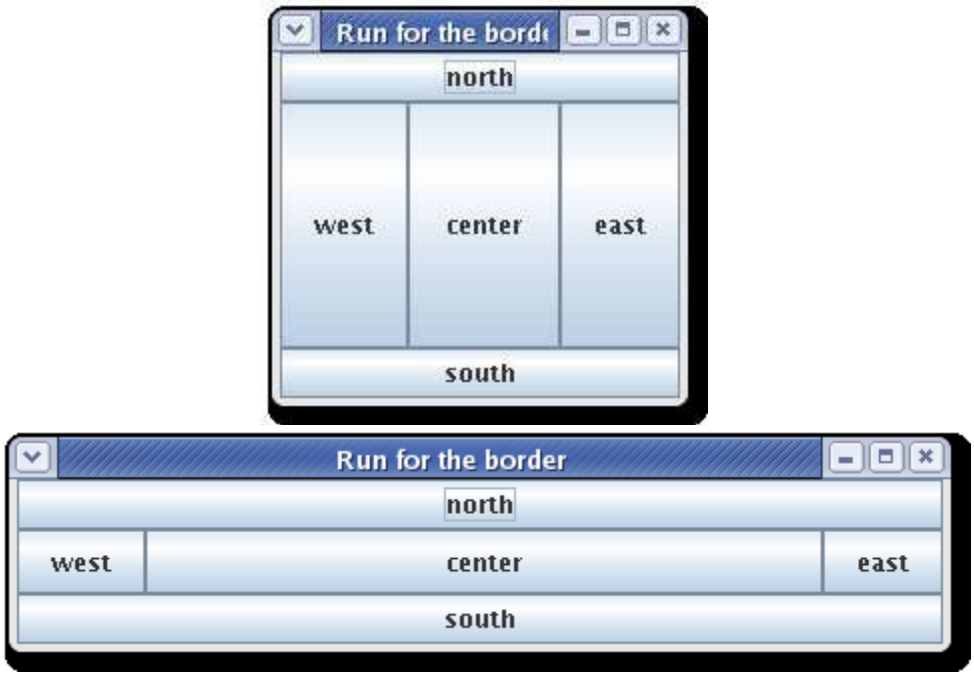


The third and final layout manager introduced in this section is named BorderLayout. BorderLayout divides the frame into five sections: north, south, west, east, and center. When adding components to a frame using a BorderLayout, you pass two parameters: the component to add, and a constant representing which region to place it.

The constants are named BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.WEST, BorderLayout.EAST, and BorderLayout.CENTER respectively. (If you do not pass a second parameter, the component will be placed in the center.)

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class BorderLayoutExample {
5      public static void main(String[] args) {
6          JFrame frame = new JFrame();
7          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8          frame.setSize(new Dimension(210, 200));
9          frame.setTitle("Run for the border");
10
11         frame.setLayout(new BorderLayout());
12         frame.add(new JButton("north"), BorderLayout.NORTH);
13         frame.add(new JButton("south"), BorderLayout.SOUTH);
14         frame.add(new JButton("west"), BorderLayout.WEST);
15         frame.add(new JButton("east"), BorderLayout.EAST);
16         frame.add(new JButton("center"), BorderLayout.CENTER);
17
18         frame.setVisible(true);
19     }
20 }
```

Here's the graphical output, both before and after resizing.



Notice the stretching behavior. The north and south buttons stretch horizontally but not vertically. The west and east buttons stretch vertically but not horizontally. The center button gets all remaining space. We don't have to place a component in all 5 regions; if we leave a region such as SOUTH empty, the CENTER consumes its space.

Here is a quick summary of the stretching and resizing behavior of all layout managers in this section:

- `FlowLayout`: Does not stretch components. Wraps to next line if necessary.
- `GridLayout`: Stretches all components in both dimensions to make them equal size at all times.

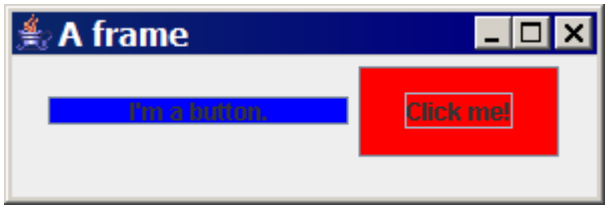
- `BorderLayout`: Stretches NORTH, SOUTH regions horizontally but not vertically. Stretches WEST, EAST regions vertically but not horizontally. Stretches CENTER region in both dimension to fill all remaining space not claimed by the other four regions.

Every graphical component has a property for its *preferred size*. Layouts like FlowLayout allow components to appear on the screen at their preferred sizes, while layouts like GridLayout ignore the preferred size. If for some reason you want to change the component's preferred size, you may set the preferred size property by calling its setPreferredSize(Dimension) method.

For example, an earlier program from a previous section created two buttons named button1 and button2 and set background colors for them. The following short modification gives the two buttons different preferred sizes.

```
button1.setPreferredSize(new Dimension(150, 14));
button2.setPreferredSize(new Dimension(100, 45));
```

The frame's FlowLayout respects the buttons' preferred sizes, leading to the following onscreen appearance:



SpringLayout

The layout managers in the previous section are somewhat simple, but they lack flexibility and power. Another layout manager named SpringLayout can be used to lay out components in a frame by linking edges of components. We don't specify the sizes of the components, but instead we specify their positions relative to edges of other components or edges of the overall frame. SpringLayout is more complicated than the others previously shown, but it is also more powerful.

We start by creating a SpringLayout object and setting our frame to use that layout. We'll save the SpringLayout into a variable, because we'll need to call methods on it in a moment:

```
SpringLayout layout = new SpringLayout();
frame.setLayout(layout);
```

The next step is that after our components are created and added to the frame, we specify constraints on their edges. Constraints are specified by calling the `putConstraint` method on the SpringLayout object. We pass five parameters when placing a constraint: a component and its edge (one of SpringLayout.WEST, SpringLayout.EAST, SpringLayout.NORTH, or SpringLayout.SOUTH), an int specifying the number of pixels of gap, and another component and its edge.

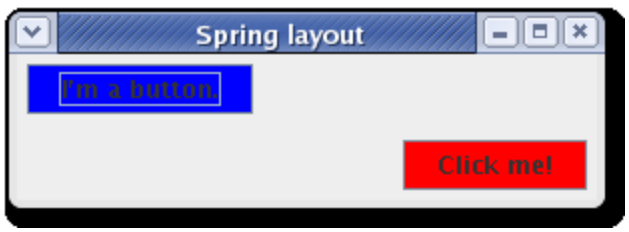
For example, if you have two JButton objects named button1 and button2, the following code places button2 exactly five pixels to the right of button1 by linking button2's west edge to button1's east edge.

```
layout.putConstraint(SpringLayout.WEST, button2, 5, SpringLayout.EAST, button1);
```

In the last example, we had two buttons. Let's put the first one in the top-left corner of the frame and the second in the bottom-right corner. To specify the frame itself as one of the edges to link, we must write `.getContentPane()` to specify that we want only the inner area that holds components, not the entire frame including its title and borders.

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class SpringLayoutExample1 {
5      public static void main(String[] args) {
6          JFrame frame = new JFrame();
7          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8          frame.setSize(new Dimension(300, 100));
9          frame.setTitle("Spring layout");
10
11         SpringLayout layout = new SpringLayout();
12         frame.setLayout(layout);
13
14         JButton button1 = new JButton();
15         button1.setBackground(Color.BLUE);
16         button1.setText("I'm a button.");
17         frame.add(button1);
18
19         JButton button2 = new JButton();
20         button2.setBackground(Color.RED);
21         button2.setText("Click me!");
22         frame.add(button2);
23
24         // keep button 1 at (5, 5)
25         layout.putConstraint(SpringLayout.WEST, button1, 5,
26                              SpringLayout.WEST, frame.getContentPane());
27         layout.putConstraint(SpringLayout.NORTH, button1, 5,
28                              SpringLayout.NORTH, frame.getContentPane());
29
30         // keep button 2's bottom-right at (width-5, height-5)
31         layout.putConstraint(SpringLayout.EAST, button2, -5,
32                              SpringLayout.EAST, frame.getContentPane());
33         layout.putConstraint(SpringLayout.SOUTH, button2, -5,
34                              SpringLayout.SOUTH, frame.getContentPane());
35
36         frame.setVisible(true);
37     }
38 }
```

When this program runs, it produces the following graphical output. The second button remains at the bottom-right corner when the window is resized:



Composite Layout

It's possible to get good component positioning without SpringLayout, using only the three simpler layout managers shown before. The trick is that layout managers may be layered on top of one another to produce combined effects. This layering is called a *composite layout*.

> **Composite Layout**
>
> A layered layout using several layout managers in different nested containers.

To create a composite layout, we must learn about containers and the JPanel class. A JFrame contains an object called a *content pane* which contains graphical components. When we've been adding buttons to our JFrame, we've really been adding to its content pane container. It's possible to construct additional containers and use them to enhance our layout. The type of object that we use as a container is named JPanel.

A JPanel object is a container to which we can add graphical components such as buttons. The usefulness of a JPanel is that it can have its own layout manager. Now only the components within that container will use that layout. By creating several containers with different layouts, we can have a fine-grained control over how each group of components behaves.

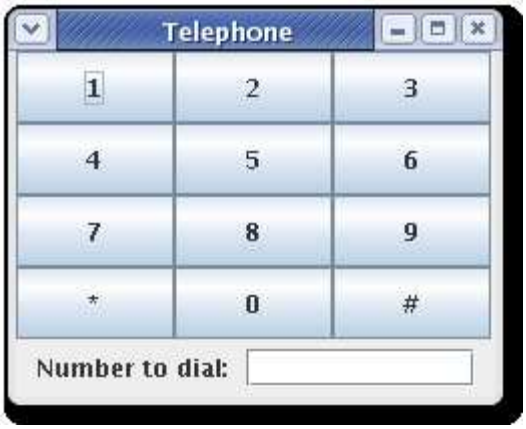
JPanels can be constructed with no parameters or with an initial layout manager to use. Once constructed, a panel can hold components that are added to it.

```
JPanel panel = new JPanel(new FlowLayout());
panel.add(new JButton("button 1"));
panel.add(new JButton("button 2"));
```

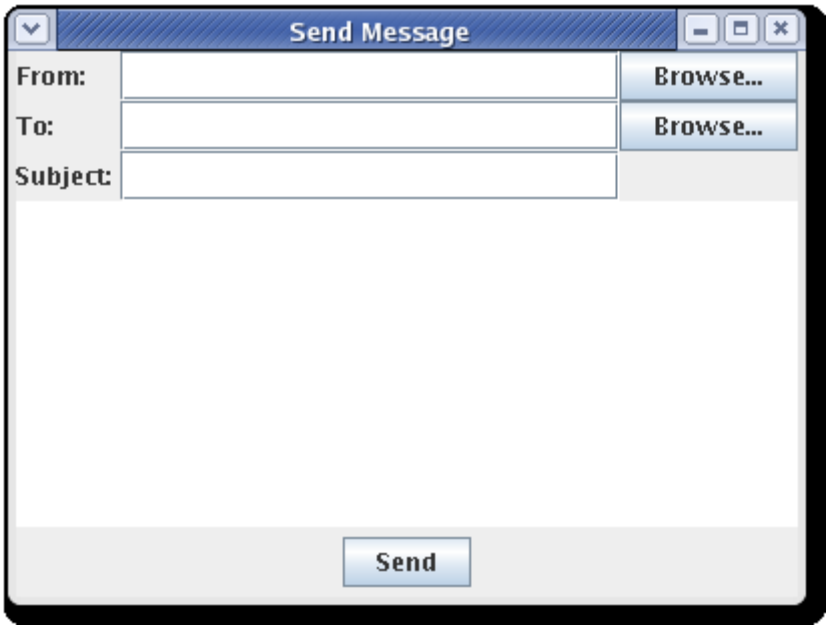
A very common strategy is to make the JFrame use a BorderLayout, then add smaller JPanel containers to some or all of the 5 regions of the frame. For example, the following code produces a window that looks like a telephone keypad:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class Telephone {
5      public static void main(String[] args) {
6          JFrame frame = new JFrame();
7          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8          frame.setSize(new Dimension(250, 200));
9          frame.setTitle("Telephone");
10
11         frame.setLayout(new BorderLayout());
12
13         // the main phone buttons
14         JPanel centerPanel = new JPanel(new GridLayout(4, 3));
15         for (int i = 1; i <= 9; i++) {
16             centerPanel.add(new JButton("" + i));
17         }
18         centerPanel.add(new JButton("*"));
19         centerPanel.add(new JButton("0"));
20         centerPanel.add(new JButton("#"));
21         frame.add(centerPanel, BorderLayout.CENTER);
22
23         // south status panel
24         JPanel southPanel = new JPanel(new FlowLayout());
25         southPanel.add(new JLabel("Number to dial: "));
26         southPanel.add(new JTextField(10));
27         frame.add(southPanel, BorderLayout.SOUTH);
28
29         frame.setVisible(true);
30     }
31 }
```

The graphical output is the following:



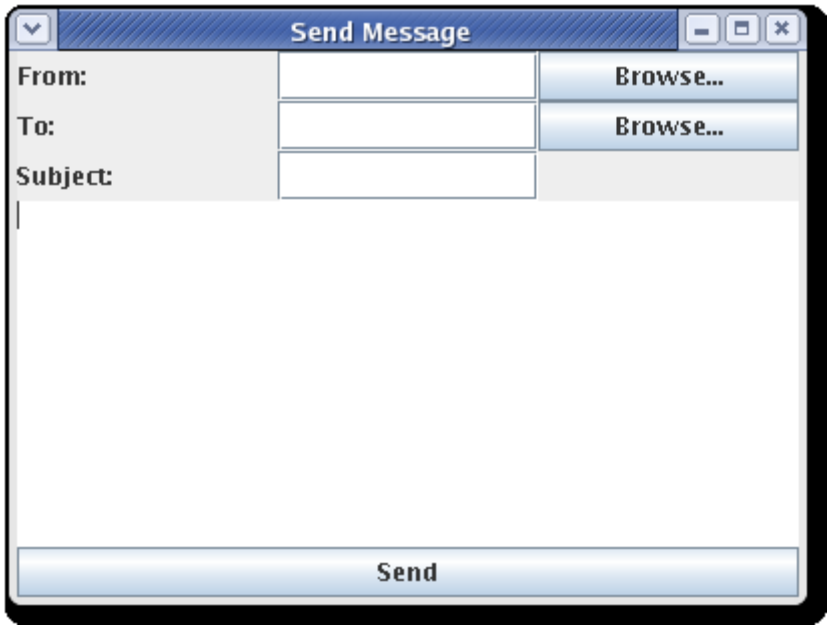
It can be very tricky to get a composite layout to look just right. It helps to practice with lots of examples and carefully consider the stretching behaviors of the various layouts. For example, how would we create a window that looks like this?



The first stab at this would be to notice that there appear to be three rows of components in the top of the window, so we could try to make a panel with a 3x3 grid layout and place the various components into it. We'd then add this to the frame, along with the central text area and Send button.

```
// This code is not quite right...
frame.setLayout(new BorderLayout());

JPanel north = new JPanel(new GridLayout(3, 3));
north.add(new JLabel("From: "));
north.add(new JTextField());
north.add(new JButton("Browse..."));
north.add(new JLabel("To: "));
north.add(new JTextField());
north.add(new JButton("Browse..."));
north.add(new JLabel("Subject: "));
north.add(new JTextField());

frame.add(north, BorderLayout.NORTH);
frame.add(new JTextArea(), BorderLayout.CENTER);
frame.add(new JButton("Send"), BorderLayout.SOUTH);
```

The following is the graphical output, which is not correct.



There are a few problems. First of all, the text labels and fields are not the proper size. This is because a grid layout forces everything inside it to take the same size. We don't want this. We want each label to be the same size as the other labels, and each text field to be the same size as the others, and so on--but we don't want the different kinds of components linked in horizontal size.

The simplest way to resolve this size problem is to create three separate JPanels with grid layouts, each with 3 rows and only 1 column. Put the labels into one grid, the text fields into the second, and the buttons into the third. (Even though there are only two buttons, make the layout 3x1 to leave a blank space to match the expected output.)

To position the three grids next to each other, we'll add another layer of compositing by creating a master north JPanel to store all three grids. The labels will occupy the west, the text fields the center, and the buttons the east. This master north JPanel will be placed into the NORTH region of the frame.

Another problem is that the Send button is stretched. This is because it's in the south region of a BorderLayout, which stretches the south horizontally. To avoid this problem, we can put the Send button into a JPanel with a FlowLayout. FlowLayout doesn't stretch the components inside it, so the button won't grow to such an odd size. Now we'll add our south JPanel to the frame rather than adding the Send button directly.

Here's a correct version of the EmailMessage program that contains these corrections and produces the proper graphical output:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class EmailMessage {
5      public static void main(String[] args) {
6          JFrame frame = new JFrame();
7          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8          frame.setSize(new Dimension(400, 300));
9          frame.setTitle("Send Message");
10         frame.setLayout(new BorderLayout());
11
12         JPanel northWest = new JPanel(new GridLayout(3, 1));
13         northWest.add(new JLabel("From: "));
14         northWest.add(new JLabel("To: "));
15         northWest.add(new JLabel("Subject: "));
16
17         JPanel northCenter = new JPanel(new GridLayout(3, 1));
18         northCenter.add(new JTextField());
19         northCenter.add(new JTextField());
20         northCenter.add(new JTextField());
21
22         JPanel northEast = new JPanel(new GridLayout(3, 1));
23         northEast.add(new JButton("Browse..."));
24         northEast.add(new JButton("Browse..."));
25
26         JPanel north = new JPanel(new BorderLayout());
27         north.add(northWest, BorderLayout.WEST);
28         north.add(northCenter, BorderLayout.CENTER);
29         north.add(northEast, BorderLayout.EAST);
30
31         JPanel south = new JPanel(new FlowLayout());
32         south.add(new JButton("Send"));
33
34         frame.add(north, BorderLayout.NORTH);
35         frame.add(new JTextArea(), BorderLayout.CENTER);
36         frame.add(south, BorderLayout.SOUTH);
37
38         frame.setVisible(true);
39     }
40 }
```

14.4 Events

So far we've created user interfaces that use graphical components and look the way we want, but they are not interactive -- nothing happens when the user clicks or types on the components. In order to create useful interactive GUIs, we must understand a Java feature named events.

When the user clicks on a component, presses a key on it, moves the mouse over it, or otherwise interacts with a component, Java's GUI system creates a special type of object to represent this action. This type of object is called an *event*.

Event

An object representing a user's interaction with a GUI component, which can be handled by your programs to create interactive components.

By default, if you don't specify anything to do about an event, it goes unnoticed by your program. Therefore, nothing happens when the user clicks your buttons or types on your text fields.

We can cause a response to a particular event (such as when the user clicks on a particular button) by creating an object called a *listener*, and attaching that listener to the component of interest. The listener object contains the code we want to run when the appropriate event occurs.

Listener

An object that is notified when an event occurs and executes code to respond to that event.

Action Events and ActionListener

The first classes we'll use for handling events in Java are named ActionEvent and ActionListener. (Both of these classes are in the java.awt.event package, so we'll need to import that.)

```
import java.awt.event.*;  // for action events
```

An action event is a fairly general event type that occurs when the user interacts with many standard components, such as clicking on a button or pressing Enter on a TextField. There are other types of events for more specific actions, such as MouseEvent, KeyEvent, and WindowEvent.

Many Java Swing components have a method named addActionListener that accepts a parameter of type ActionListener. An ActionListener is an object that can be notified when events occur and can respond to those events.

Useful Methods of Components

`public void addActionListener(ActionListener listener)`
Used to attach an ActionListener to hear action events.

ActionListener is actually not a class but an interface. To listen to an event, we write a class that implements the ActionListener interface, and we specify in that class how we want to respond to the event. Then we attach an object of our listener class to the appropriate component.

The ActionListener interface contains only the following method:

```
public void actionPerformed(ActionEvent event)
```

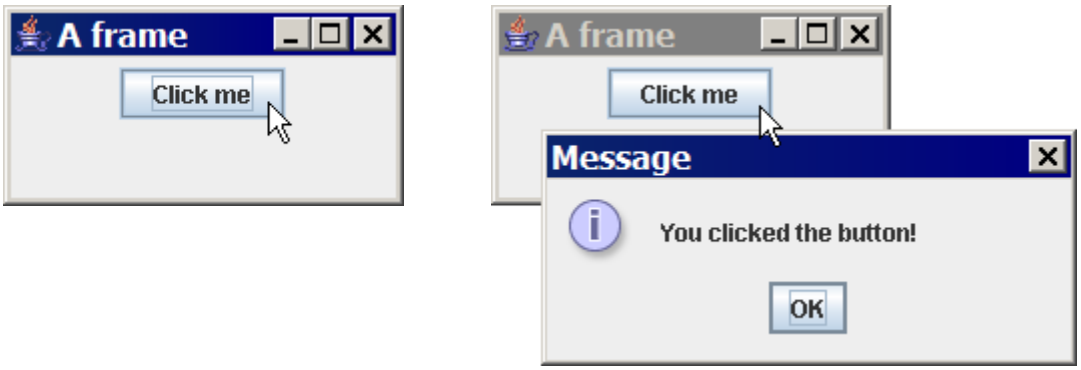
Here is a simple example of an `ActionListener` class that responds to an event by displaying a message dialog box on the screen. For now we will ignore the ActionEvent parameter and not use it in our code inside the actionPerformed method.

```
// Responds to a button click event by displaying a message dialog box.
public class MessageListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        JOptionPane.showMessageDialog(null, "You clicked the button!");
    }
}
```

Now that we've created this class, we can attach MessageListener objects to buttons. This will cause a message to pop up whenever that button is clicked.

```
JButton button = new JButton("Click me");
MessageListener listener = new MessageListener();
button.addActionListener(listener);
```

Here is the result when the program is executed, and when the user clicks on the button:



To summarize, here are the necessary steps to handle an event in Java:

1. Write a class that implements ActionListener
2. Place the code to handle the event into its actionPerformed method
3. Attach an object of your listener class to the component you want to listen to

More Sophisticated ActionEvents

The ActionEvent object passed as the parameter to an ActionListener's actionPerformed method can be useful, depending on what response we want to occur. The ActionEvent has methods that we can call to find out more about the event that occurred -- what component was clicked, and so on.

Useful Methods of ActionEvent objects

`public String getActionCommand()` Returns a String representing the event that occurred. For example, if the component in question is a JButton, this method returns the button's text. If it is a JTextField, this method returns the text the user has typed into the text field.
`public Object getSource()` Returns a reference to the component that generated the event. Since the return type is written as Object to be general, you'll have to cast it to the type of component you expect.

The following `ActionListener` could be attached to a JButton. It would alternate the button's background color between red and blue each time the user clicked the button.

```
// Responds to a button click event by displaying a message dialog box.
public class MessageListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        JButton button = (JButton) event.getSource();
        if (button.getBackground() == Color.RED) {
            button.setBackground(Color.BLUE);
        } else {
            button.setBackground(Color.RED);
        }
    }
}
```

The following `ActionListener` could be attached to a JTextField. It would capitalize the text in the text field when the user presses Enter in the field.

```
// Responds to a text field Enter keypress by capitalizing its text
public class CapitalizeListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        JTextField field = (JTextField) event.getSource();
        field.setText(field.getText().toUpperCase());
    }
}
```

A Larger GUI Example with Events: Credit Card GUI

In larger graphical programs, your responses to an event that occurs in one component will affect another component. In this section, we'll explore an example program that needs event responses that affect multiple components.

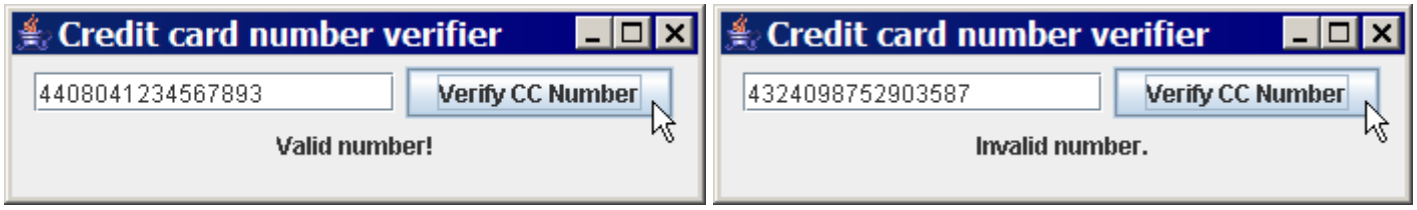
You may not know that credit card numbers contain several pieces of information for performing validity tests. For example, Visa card numbers always begin with 4, and a valid Visa card number also passes a digit-sum test known as the Luhn checksum algorithm. Luhn's algorithm states that if you sum the digits of the number in a certain way, the total sum must be a multiple of 10 for a valid Visa number. Systems that accept credit cards perform a Luhn test before contacting the credit card company for final verification. This lets the company weed out many fake or incorrect credit card numbers.

The algorithm for summing the digits is the following. For digits at even indexes (the 0th digit, 2nd digit, etc.), simply add that digit to the cumulative sum. For digits at odd indexes (index 1, 3, etc.), double the digit's value, then if that doubled value is more than 10, add its digits together to make a number that is smaller than 10, then add this result into the sum.

Here is an example credit card number that passes the Luhn algorithm. Notice how the number 5 at index 10 is doubled to 10 which becomes (1+0), how the number 7 at index 12 is doubled to 14 which becomes (1+4), and so on.

```
4408 0412 3456 7893

TOTAL = (8)+4+(0)+8   + (0)+4+(2)+2 +
        (6)+4+(1+0)+6 + (1+4)+8+(1+8)+3
TOTAL = 70

70 % 10 == 0, therefore this card number is valid
```

We could write a GUI where the user could type in a credit card number, press a button to verify, then receive a message stating whether the number was valid. The GUI might look like this:



However, let's think about the event response code for a second. Clicking the "Verify CC Number" button would cause the action event, so we'd put an ActionListener on the button. But in the code for the actionPerformed method, we'd need read the text from the top-left text field, decide whether the number was valid, and use that information to set the text of the bottom JLabel. The GUIs we've written so far were not built to allow so much interaction between components.

To make a GUI that does allow access to each component from your listeners, make your GUI itself into an object. Declare your components as data fields inside that object, and initialize them in the GUI's constructor. The following code demonstrates this idea. We'll also write an ActionListener class named VerifyListener and attach the listener to our Verify button. We'll implement the VerifyListener in a moment.

```
public class CreditCardGUI {
    private JFrame frame;
    private JTextField numberField;
    private JLabel validLabel;
    private JButton verifyButton;

    public CreditCardGUI() {
        frame = new JFrame("Credit card number verifier");
        numberField = new JTextField(16);
        validLabel = new JLabel("not yet verified");
        verifyButton = new JButton("Verify CC Number");
        // ... (some code omitted)
        frame.setVisible(true);
    }
```

To run the program, you can include a main method in your GUI class that calls the constructor to create your GUI object.

```
public static void main(String[] args) {
    new CreditCardGUI();
}
```

This is a bit of an odd line of code, calling a constructor as a statement. Normally we only call constructors when declaring or initializing reference variables, but in this case we just want to create the CreditCardGUI object and run its constructor. This will change us from the static context of the main method into the the context of the CreditCardGUI object. The main method doesn't need a reference to it again after this line, so we simply call the constructor as the only statement in the main method.

Now let's write the VerifyListener to listen to the action events on the Verify button. As we said earlier, this listener class will need to be able to communicate with the numberField JTextField and the validLabel JLabel. To make this easier, we'll use a Java feature called an inner class. An inner

class is a class that is declared inside of another class. The inner class can only be used in conjunction with the outer class, and it can access and use all of the data fields and methods from the outer class. Inner classes are especially useful for complicated event listeners that need to manipulate several GUI components.

Inner class

A Java class declared inside of another class. Inner classes can access all data fields and methods of the enclosing outer class.

Inner classes are declared outside any methods of the outer class, at the same indentation level as its methods. Inner classes use the same syntax as regular classes, except that they are bound to the enclosing object of their outer class that created them, and therefore they can access the data fields and methods of that outer object.

Assuming we had a method `public boolean isValidCreditCardNumber(String number)` that told us whether a given String represents a valid Visa card number, we could write an `ActionListener` as an inner class, that called this method and displayed the result in the text label. It is legal for the VerifyListener's code to refer to those component data fields from the GUI, because the VerifyListener class is an inner class inside the GUI. This is the main reason we chose to make the listener an inner class.

```
class VerifyListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        String text = numberField.getText();
        if (isValidCreditCardNumber(text)) {
            validLabel.setText("Valid number!");
        } else {
            validLabel.setText("Invalid number.");
        }
    }
}
```

The following is the code for the complete CreditCardGUI program.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Example valid numbers to test: 4111111111111111, 4408041234567893
public class CreditCardGUI {
    public static void main(String[] args) {
        new CreditCardGUI();
    }

    private JFrame frame;
    private JTextField numberField;
    private JLabel validLabel;
    private JButton verifyButton;

    public CreditCardGUI() {
        // set up components
        frame = new JFrame("Credit card number verifier");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(350, 100));
        numberField = new JTextField(16);
        validLabel = new JLabel("not yet verified");
        verifyButton = new JButton("Verify CC Number");

        // event listeners
        verifyButton.addActionListener(new VerifyListener());

        // layout
        frame.setLayout(new FlowLayout());
        frame.add(numberField);
        frame.add(verifyButton);
        frame.add(validLabel);
        frame.setVisible(true);
    }

    // Returns whether the given string is a valid Visa card number
    // according to the Luhn checksum algorithm.
    public boolean isValidCreditCardNumber(String text) {
        if (!text.startsWith("4")) {
            return false;
        }

        // add all of the digits
        int sum = 0;
        for (int i = 0; i < text.length(); i++) {
            int digit = Integer.valueOf(text.substring(i, i + 1));
            if (i % 2 == 0) {  // double every other number, add digits
                digit *= 2;
                sum += (digit / 10) + (digit % 10);
            } else {
                sum += digit;
            }
        }
        // valid numbers add up to a multiple of 10
        return (sum % 10 == 0);
    }

    // Sets the label's text to show whether the credit card number is valid.
    public class VerifyListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
```

```
61              String text = numberField.getText();
62              if (isValidCreditCardNumber(text)) {
63                  validLabel.setText("Valid number!");
64              } else {
65                  validLabel.setText("Invalid number.");
66              }
67          }
68      }
69  }
```

Mouse Events

When we want to listen to mouse clicks or movement, we use another type of listener named a MouseInputAdapter. This class resides in the javax.swing.event package, so you'll need to import it.

```
import javax.swing.event.*;  // for mouse events
```

While you write an `ActionListener` by implementing an interface, you write a mouse listener by extending a pre-existing class. That class is named MouseInputAdapter, and it has the following methods:

Useful Methods of MouseInputAdapter objects

`public void mouseClicked(MouseEvent event)` Invoked when the mouse button has been clicked (pressed and released) on a component.
`public void mouseDragged(MouseEvent event)` Invoked when a mouse button is pressed on a component and then dragged.
`public void mouseEntered(MouseEvent event)` Invoked when the mouse enters a component.
`public void mouseExited(MouseEvent event)` Invoked when the mouse exits a component.
`public void mouseMoved(MouseEvent event)` Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.
`public void mousePressed(MouseEvent event)` Invoked when a mouse button has been pressed on a component.
`public void mouseReleased(MouseEvent event)` Invoked when a mouse button has been released on a component.

There are quite a few methods, and you probably won't want to implement them all. The default version of these methods in MouseInputAdapter is an empty method that does nothing. So for example, to make a class that only responds to a mouseEntered event, extend the MouseInputAdapter class and override only the mouseEntered method.

```
public class MovementListener extends MouseInputAdapter {
    public void mouseEntered(MouseEvent event) {
        JOptionPane.showMessageDialog(null, "Mouse entered!");
    }
}
```

Unfortunately, Java was designed in such a way that we need to learn about two uses of mouse listeners. Java's designers decided to separate the concept of mouse button clicks from the idea of mouse movement, creating two separate interfaces named MouseListener and MouseMotionListener to represent these ideas respectively. Later the MouseInputListener and MouseInputAdapter were added to address this by merging the two listeners, but GUI components still need the listener to be attached in two separate ways for it to work.

The result of this is that if we wish to hear about mouse button presses, such as the mousePressed or mouseReleased methods, we call the addMouseListener method on the component. If we wish to hear about mouse motion, such as the mouseMoved or mouseDragged methods, we call the addMouse**Motion**Listener method on the component. If we want to hear all the events, we can add the mouse input adapter in both ways. (This is what we'll do in the examples in this chapter.)

Useful Methods of Components

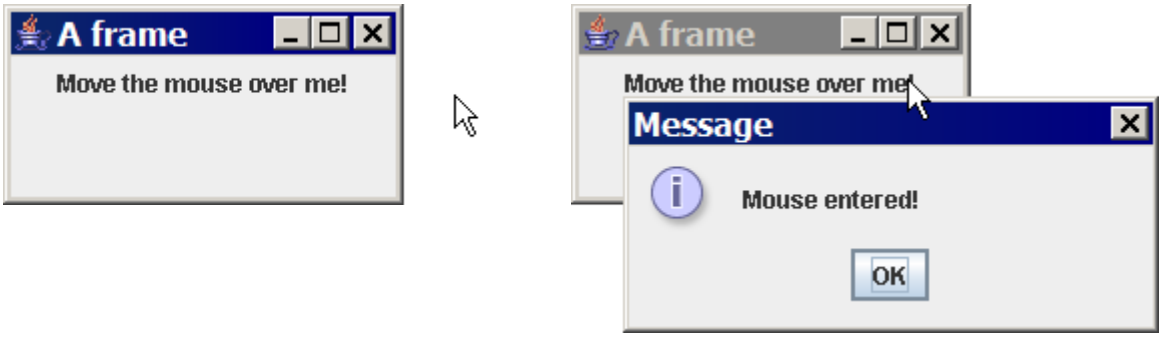
`public void addMouseListener(MouseListener listener)` Used to attach a MouseInputAdapter to hear mouse enter, exit, press, release, and click events.
`public void addMouseMotionListener(MouseMotionListener listener)` Used to attach a MouseInputAdapter to hear mouse move and drag events.

Here is an entire program that uses the listener to respond to moving the mouse over the JLabel:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class MouseGUI {
5      public static void main(String[] args) {
6          JFrame frame = new JFrame();
7          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8          frame.setLayout(new FlowLayout());
9          frame.setSize(new Dimension(200, 100));
10         frame.setTitle("A frame");
11
12         JLabel label = new JLabel();
13         label.setText("Move the mouse over me!");
14         frame.add(label);
15
16         MovementListener mListener = new MovementListener();
17         label.addMouseListener(mListener);
18         label.addMouseMotionListener(mListener);
19
20         frame.setVisible(true);
21     }
22 }
```

```
1  import java.awt.event.*;
2  import javax.swing.*;
3  import javax.swing.event.*;
4
5  public class MovementListener extends MouseInputAdapter {
6      public void mouseEntered(MouseEvent event) {
7          JOptionPane.showMessageDialog(null, "Mouse entered!");
8      }
9  }
```

The program produces the following graphical output, shown both before and after the user moves the mouse onto the JLabel:



Just like we saw in the previous section with ActionEvents, there are useful pieces of information stored in the MouseEvent parameter. Here is a partial list:

Useful Methods of MouseEvent objects

`public int getButton()` Returns the number of the mouse button that was pressed or released (1 for the left button, 2 for the right button, and so on).
`public int getClickCount()` Returns the number of times the user clicked the button; useful for detecting double-clicks.
`public Point getPoint()` Returns the (x, y) point where the mouse event occurred.
`public int getX()` Returns the x-coordinate where the mouse event occurred.
`public int getY()` Returns the y-coordinate where the mouse event occurred.
`public Object getSource()` Returns a reference to the component that generated the event. Since the return type is written as Object to be general, you'll have to cast it to the type of component you expect.

For example, the following mouse listener could be attached to any component, even the JFrame itself. It would make a message appear any time the user pressed the mouse button:

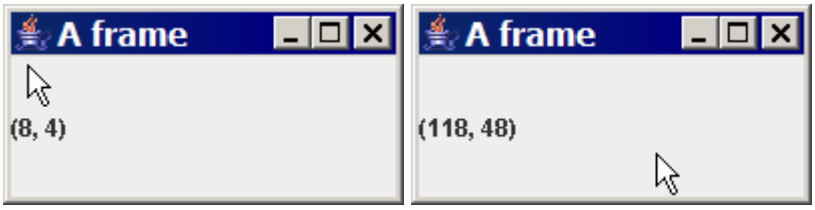
```
// Responds to a mouse click by showing a message of where the user clicked
public class ClickListener extends MouseInputAdapter {
    public void mousePressed(MouseEvent event) {
        JOptionPane.showMessageDialog(null, "Mouse pressed at position ("
            + event.getX() + ", " + event.getY() + ")");
    }
}
```

Here is an example program that uses a MouseInputAdapter to set a JLabel's text to show the mouse's position as it moves over the JLabel.

```
import java.awt.*;
import javax.swing.*;

public class MousePointGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(200, 100));
        frame.setTitle("A frame");

        JLabel label = new JLabel();
        label.setText("Move the mouse over me!");
        frame.add(label);

        PointListener mListener = new PointListener();
        label.addMouseListener(mListener);
        label.addMouseMotionListener(mListener);

        frame.setVisible(true);
    }
}
```

```
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class PointListener extends MouseInputAdapter {
    public void mouseMoved(MouseEvent event) {
        JLabel label = (JLabel) event.getSource();
        label.setText("(" + event.getX() + ", " + event.getY() + ")");
    }
}
```

The program produces the following graphical output, shown after the user moves the mouse onto several points on the JLabel:



14.5 2D Graphics

Earlier in this textbook, we introduced a graphical class named DrawingPanel. DrawingPanel was kept simple so that you did not need to learn a lot of details about graphical user interfaces. Now that we're starting to uncover those details, we'll examine how to do our own 2D graphics manually. This will lead us to the point where we could implement DrawingPanel by ourselves, if we so chose.

Drawing Onto Panels

Earlier in this chapter we saw the JPanel component, which we used as a container for laying out components. JPanels have one other important function: they serve as surfaces onto which we can draw. The JPanel class has a method named paintComponent that draws the panel on the screen. By default, this method draws nothing, so the panel is transparent. If we wanted to change this default behavior, how could we do it?

If you thought of using inheritance, you're on the right track. We can extend JPanel and override the paintComponent method if we want to make a panel that has shapes drawn on it. Here is the signature of the paintComponent method we must override:

```
public void paintComponent(Graphics g)
```

This parameter, Graphics g, should look familiar to you if you used DrawingPanel from the Supplement 3G earlier in the textbook. Graphics is an object in Java's java.awt package that houses methods for drawing shapes, lines, and images onto a surface. Think of the Graphics as a pen and the JPanel as a sheet of paper.

There is one quirk when extending JPanel and writing a paintComponent method. Remember that when you override a method, you replace the superclass method's previous functionality. We don't want to lose the behavior of paintComponent from JPanel, because it does important things on the interior of the panel. We just want to add to it. Therefore, the first thing you should do when overriding paintComponent is to call super.paintComponent, and pass it your same Graphics g as its parameter. (If you don't, you can get strange ghosty afterimages when you drag or resize your window.)

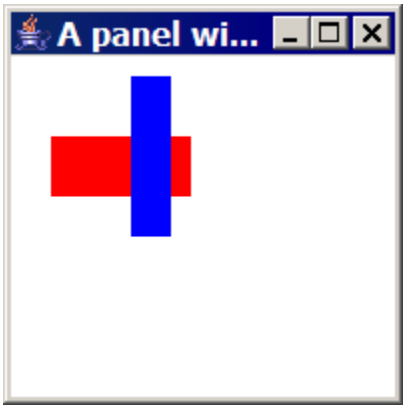
Here's a small class that represents a panel with two rectangles drawn on it:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class RectPanel extends JPanel {
5      public void paintComponent(Graphics g) {
6          super.paintComponent(g);  // call JPanel's original version
7          g.setColor(Color.RED);
8          g.fillRect(20, 40, 70, 30);
9          g.setColor(Color.BLUE);
10         g.fillRect(60, 10, 20, 80);
11     }
12 }
```

Now, in your GUI class, you can create a RectPanel object and add it to your JFrame. It will appear on screen and will have the shapes drawn on it. Here's the example client code that uses our RectPanel:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class UseRectPanel {
5      public static void main(String[] args) {
6          JFrame frame = new JFrame();
7          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8          frame.setSize(200, 200);
9          frame.setTitle("A panel with rectangles");
10
11         RectPanel panel = new RectPanel();
12         panel.setBackground(Color.WHITE);
13         frame.add(panel);
14
15         frame.setVisible(true);
16     }
17 }
```

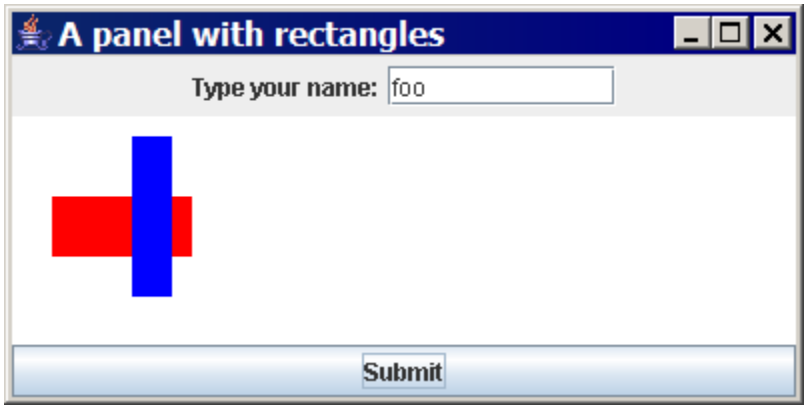
The program produces the following graphical output:



You might ask yourself, why not just use a DrawingPanel rather than going to all this trouble? DrawingPanel is a good tool for simple drawing, but there are many things it can't do. For example, we can't make a DrawingPanel part of a larger GUI with other components. The following modified code for the previous client program achieves a mixture of components that would be impossible to do with DrawingPanel:

```
public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(400, 200);
    frame.setTitle("A panel with rectangles");
    frame.setLayout(new BorderLayout());

    JPanel north = new JPanel(new FlowLayout());
    north.add(new JLabel("Type your name:"));
    north.add(new JTextField(10));
    frame.add(north, BorderLayout.NORTH);

    frame.add(new JButton("Submit"), BorderLayout.SOUTH);

    RectPanel panel = new RectPanel();
    panel.setBackground(Color.WHITE);
    frame.add(panel, BorderLayout.CENTER);

    frame.setVisible(true);
}
```

The program produces the following graphical output:



For a more complete description of 2D graphical methods and objects, see Supplement 3G's discussion of DrawingPanel and Graphics.

Simple Animation with Timers

A JPanel doesn't have to show a simple static drawing. A panel can be animated to show a moving picture, through the use of an object named a Timer. A Timer is an object that, once started, fires an action event at regular intervals. You supply the timer with an `ActionListener` to use and a delay time in milliseconds, and it does the rest. We'll need two things in order to do animation:

1. An ActionListener object that somehow updates the way the panel will draw itself
2. A Timer object to invoke that ActionListener at regular intervals, causing the panel to animate

Let's modify our RectPanel from the previous section so that its rectangles move on the panel. To make them move, we must store their positions as data fields in the panel object. We'll change the data fields' values as the program is running, then redraw the rectangles, which will make it look like they're moving. We'll store the desired change-in-x and change-in-y as variables named dx and dy, respectively.

```
// Initial version -- will change in the following section.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class AnimatedRectPanel extends JPanel {
    private Point p1;  // location of first rectangle
    private Point p2;  // location of second rectangle
    private int dx;    // amount by which to move horizontally
    private int dy;    // amount by which to move vertically

    public AnimatedRectPanel() {
        p1 = new Point(20, 40);
        p2 = new Point(60, 10);
        dx = 5;
        dy = 5;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);  // call JPanel's original version
        g.setColor(Color.RED);
        g.fillRect(p1.x, p1.y, 70, 30);
        g.setColor(Color.BLUE);
        g.fillRect(p2.x, p2.y, 20, 80);
    }
}
```

Let's make a method named move that shifts each rectangle's position slightly. Our `ActionListener` can call this move method repeatedly to animate the panel. Let's decide that the first rectangle will move horizontally back and forth on the panel, "bouncing" back and forth off the panel's left and right edges. The second rectangle will move vertically, bouncing off the top and bottom edges. The most straightforward thing to do in our ActionListener is to adjust the two points' coordinates.

```
// Initial version -- doesn't turn around.
public void move() {
    p1.x += dx;
    p2.y += dy;
}
```

However, since the rectangles are supposed to bounce off the edges, we need to check to see if a rectangle has hit either edge. We'll know the rectangle has hit the edge when its left edge has dropped to 0, or when its right edge (the left edge plus the width) has passed the panel's width. The calculation is similar for the y, except that we'll look at the panel's height.

```
// adjusts the position of both rectangles
public void move() {
    p1.x += dx;
    p2.y += dy;
    if (p1.x <= 0 || p1.x + 70 >= getWidth()) {
        dx = -dx;  // rectangle 1 has hit left/right edge
    }
    if (p2.y <= 0 || p2.y + 80 >= getHeight()) {
        dy = -dy;  // rectangle 2 has hit top/bottom edge
    }
}
```

The next step is to create an ActionListener that moves the panel each time it is invoked. We'll make this class accept an AnimatedRectPanel as a parameter to its constructor and call move() on that panel when its `ActionListener` is invoked.

There's one last thing we need to know about animating JPanels. When we change the way the JPanel should be drawn, we have to call a method named `repaint` to instruct the JPanel to draw itself again. If we don't call repaint, we won't see any change on the screen at all; this is a common bug.

Here's the complete ActionListener that moves and repaints the panel. It is implemented as an inner class inside AnimatedRectPanel, so that it can call the move and repaint methods of that object.

```
// redraws the rectangle panel at timed intervals
public class RectangleMover implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        move();
        repaint();
    }
}
```

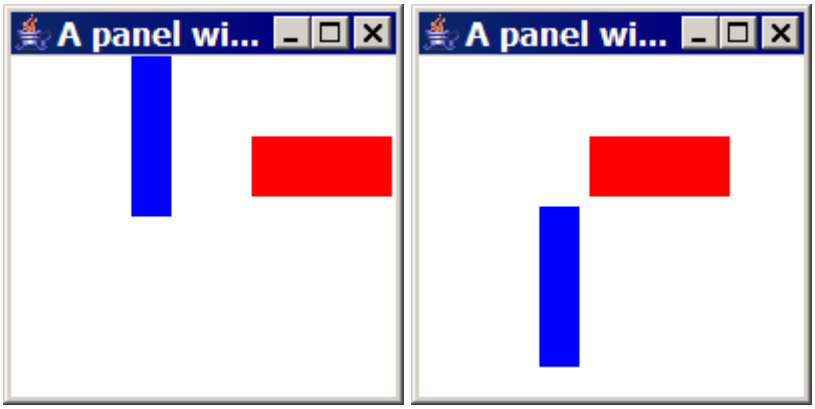
Once our ActionListener is complete, we have to create and start the timer that invokes the ActionListener. Let's add code for this to the constructor for the AnimatedRectPanel.

```
// set up the timer to animate the motion of the rectangles
RectangleMover mover = new RectangleMover();
Timer time = new Timer(100, mover);  // update every 100 ms
time.start();
```

Here's the complete AnimatedRectPanel class that contains an ActionListener that incorporates all of these features.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class AnimatedRectPanel extends JPanel {
6      private Point p1;  // location of first rectangle
7      private Point p2;  // location of second rectangle
8      private int dx;    // amount by which to change position horizontally
9      private int dy;    // amount by which to change position vertically
10
11     public AnimatedRectPanel() {
12         p1 = new Point(20, 40);
13         p2 = new Point(60, 10);
14         dx = 5;
15         dy = 5;
16
17         // set up the timer to animate the motion of the rectangles
18         RectangleMover mover = new RectangleMover();
19         Timer time = new Timer(100, mover);  // update every 100 ms
20         time.start();
21     }
22
23     // draws two rectangles on this panel on the screen
24     public void paintComponent(Graphics g) {
25         super.paintComponent(g);  // call JPanel's original version
26         g.setColor(Color.RED);
27         g.fillRect(p1.x, p1.y, 70, 30);
28         g.setColor(Color.BLUE);
29         g.fillRect(p2.x, p2.y, 20, 80);
30     }
31
32     // adjusts the position of both rectangles
33     public void move() {
34         p1.x += dx;
35         p2.y += dy;
36         if (p1.x <= 0 || p1.x + 70 >= getWidth()) {
37             dx = -dx;  // rectangle 1 has hit left/right edge
38         }
39         if (p2.y <= 0 || p2.y + 80 >= getHeight()) {
40             dy = -dy;  // rectangle 2 has hit top/bottom edge
41         }
42     }
43
44     // redraws the rectangle panel at timed intervals
45     public class RectangleMover implements ActionListener {
46         public void actionPerformed(ActionEvent event) {
47             move();
48             repaint();
49         }
50     }
51 }
```

The client code to display this new animated rectangle panel looks identical to the UseRectPanel class shown before, except the name AnimatedRectPanel appears in place of RectPanel. The graphical window produced as output animates, just as we intended.



Here is a list of the methods you may use from the Timer class:

Useful Methods of Timer objects

Method	Description
`public Timer(int msDelay, ActionListener listener)`	Creates a Timer that activates the given `ActionListener` every msDelay milliseconds.
`public void start()`	tells the timer to start ticking and activating its `ActionListener`
`public void stop()`	tells the timer to stop ticking and not activate its `ActionListener`

14.6 Case Study: Demystifying DrawingPanel

In previous chapters, we have used the DrawingPanel class for simple 2D graphics. In this final section, we'll analyze the DrawingPanel in detail to understand how it works.

The start of the DrawingPanel class looks a lot like the other GUIs we have used in this chapter, declaring various graphical components as data fields. The main fields are the overall window frame, a panel on which the user can draw, and a status bar label that shows the mouse's position. The DrawingPanel also keeps a Graphics2D object as a data field, which will represent the pen used to draw on the main panel. DrawingPanel declares a constant delay which is used as the delay for a Timer.

```
public class DrawingPanel {
    public static final int DELAY = 250;  // ms delay between repaints

    private JFrame frame;          // overall window frame
    private JPanel panel;          // overall drawing surface
    private Graphics2D g2;         // graphics context for painting
    private JLabel statusBar;      // status bar showing mouse position
```

The constructor of the DrawingPanel initializes the data fields, as well as constructing a BufferedImage to serve as the persistent buffer where shapes and lines can be drawn. The image is set to be the icon of an onscreen JLabel.

```
public DrawingPanel(int width, int height) {
    // set up the empty image onto which we will draw
    BufferedImage image = new BufferedImage(width, height, TYPE_INT_ARGB);
    g2 = (Graphics2D) image.getGraphics();
    g2.setColor(Color.BLACK);

    JLabel label = new JLabel();
    label.setIcon(new ImageIcon(image));

    panel = new JPanel();
    panel.setLayout(new FlowLayout(0, 0));
    panel.setBackground(Color.WHITE);
    panel.setPreferredSize(new Dimension(width, height));
    panel.add(label);
```

There are a few quirks in the preceding code. One quirk is that we'll cast the Graphics of our BufferedImage into a Graphics2D for programs that want the extra Graphics2D features. (Graphics2D is discussed in Chapter 9.) Another quirk is that we're passing two integers as parameters to the constructor of our FlowLayout object. We've done this to specify that it not place any padding around its edges. The panel is being used solely for layout purposes, to make the onscreen image appear in the right place and at the right size.

Next, a special mouse listener is attached to the main panel, so that whenever the mouse moves on the DrawingPanel, the status bar text can be updated to show the mouse position.

```
    // the status bar that shows the mouse position
    statusBar = new JLabel(" ");

    StatusBarMouseAdapter mouse = new StatusBarMouseAdapter();
    panel.addMouseListener(mouse);
    panel.addMouseMotionListener(mouse);
```

Here is the code for the mouse listener, which is used for debugging purposes. It sets the text of the status bar to reflect the mouse's current position every time it moves:

```
class StatusBarMouseAdapter extends MouseInputAdapter {
    public void mouseMoved(MouseEvent e) {
        statusBar.setText("(" + e.getX() + ", " + e.getY() + ")");
    }
}
```

Next, the main window frame is created and shown on the screen. The last action in the DrawingPanel's constructor is to create and start a Timer, which periodically repaints the screen. This is done to ensure that every shape the user draws will appear on the screen, even if the client's code takes a few seconds to run.

```
    // set up the JFrame
    frame = new JFrame("DrawingPanel");
    frame.setResizable(false);
    frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
    frame.add(panel);
    frame.add(statusBar, SOUTH);
    frame.pack();
    frame.setVisible(true);
    frame.toFront();

    // start a repaint timer so that the screen will update
    TimerListener listener = new TimerListener();
    Timer timer = new Timer(DELAY, listener);
    timer.start();
```

The ActionListener attached to the timer simply repaints the main panel:

```
class TimerListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        panel.repaint();
    }
}
```

As you can see, the DrawingPanel largely uses concepts that have been introduced in this chapter, and it now consists of components and programming constructs that you have seen. If you wish, you could modify or enhance the functionality of DrawingPanel yourself. Here is the complete code for the DrawingPanel class:

```
 1  // The DrawingPanel class provides a simple interface for drawing persistent
 2  // images using a Graphics object.
 3
 4  import java.awt.*;
 5  import java.awt.event.*;
 6  import java.awt.image.*;
 7  import javax.swing.*;
 8  import javax.swing.event.*;
 9
10  public class DrawingPanel {
11      public static final int DELAY = 250;  // ms delay between repaints
12
13      private JFrame frame;         // overall window frame
14      private JPanel panel;         // overall drawing surface
15      private Graphics g;           // graphics context for painting
16      private JLabel statusBar;     // status bar showing mouse position
17
18      // constructs a drawing panel of given width and height enclosed in a window
19      public DrawingPanel(int width, int height) {
20          // set up the empty image onto which we will draw
21          BufferedImage image = new BufferedImage(width, height,
22                                  BufferedImage.TYPE_INT_ARGB);
23          g = image.getGraphics();
24          g.setColor(Color.BLACK);
25          JLabel label = new JLabel();
26          label.setIcon(new ImageIcon(image));
27
28                  panel = new JPanel();
29                  panel.setLayout(new FlowLayout(0, 0));
30          panel.setBackground(Color.WHITE);
```

```
31          panel.setPreferredSize(new Dimension(width, height));
32          panel.add(label);
33
34          // the status bar that shows the mouse position
35          statusBar = new JLabel(" ");
36
37          StatusBarMouseAdapter mouse = new StatusBarMouseAdapter();
38          panel.addMouseListener(mouse);
39          panel.addMouseMotionListener(mouse);
40
41          // set up the JFrame
42          frame = new JFrame("Drawing Panel");
43          frame.setResizable(false);
44          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
45          frame.add(panel);
46          frame.add(statusBar, BorderLayout.SOUTH);
47          frame.pack();
48          frame.setVisible(true);
49          frame.toFront();
50
51          // start a repaint timer so that the screen will update
52          TimerListener listener = new TimerListener();
53          Timer timer = new Timer(DELAY, listener);
54          timer.start();
55      }
56
57      // obtain the Graphics object to draw on the panel
58      public Graphics getGraphics() {
59          return g;
60      }
61
62      // set the background color of the drawing panel
63      public void setBackground(Color c) {
64          panel.setBackground(c);
65      }
66
67      // show or hide the drawing panel on the screen
68      public void setVisible(boolean visible) {
69          frame.setVisible(visible);
70      }
71
72      // makes the program pause for the given amount of time, for animation
73      public void sleep(int millis) {
74          try {
75              Thread.sleep(millis);
76          } catch (InterruptedException e) {}
77      }
78
79      // makes drawing panel become the frontmost window on the screen
80      public void toFront() {
81          frame.toFront();
82      }
83
84      // used for an internal timer that repeatedly repaints the screen
85      class TimerListener implements ActionListener {
86          public void actionPerformed(ActionEvent e) {
87              panel.repaint();
88          }
89      }
90
```

```
91      // draws the status bar text when the mouse moves
92      class StatusBarMouseAdapter extends MouseInputAdapter {
93          public void mouseMoved(MouseEvent e) {
94              statusBar.setText("(" + e.getX() + ", " + e.getY() + ")");
95          }
96      }
97  }
```

Chapter Summary

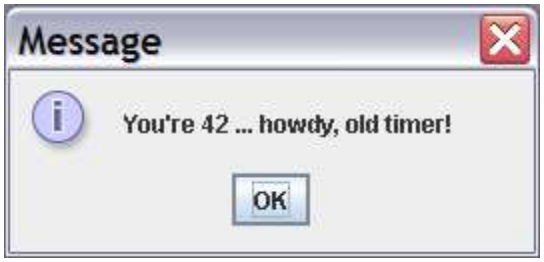
- A graphical user interface (GUI) has a window frame that contains buttons, text input fields, and other onscreen components.
- The JOptionPane class is a simple GUI class for performing graphical input and output by displaying messages or prompting for input values.
- Some of the most common graphical components are buttons (JButton), text input fields (JTextField / JTextArea), and text labels (JLabel).
- All graphical components in Java belong to a common inheritance hierarchy, so they share a common set of methods and properties, such as background color, size, and font.
- To properly position components on the screen, they must be added to a window frame (JFrame). Components are positioned by objects called layout managers. Common layout managers include BorderLayout, FlowLayout, GridLayout, and SpringLayout.
- The features of the layout managers can be combined by nesting several containers, each using a different layout manager. This is called composite layout.
- Java generates special objects named events when the user interacts with onscreen graphical components. To write an interactive GUI, you must respond to these events.
- The most common event is an ActionEvent, which is handled by writing a class that implements the ActionListener interface. You can also respond to MouseEvents, by writing a class that extends the MouseInputAdapter class.
- To draw lines and shapes, like the DrawingPanel from previous chapters does, you must write a class that extends JPanel and write a method named paintComponent. The paintComponent method specifies how to draw the panel, and it is given a parameter of type Graphics, so you can draw any colors or shapes you like in this method.

Self-Check Problems

Section 14.1: Graphical Input and Output with JOptionPane

1. What package contains Java's "Swing" GUI classes used in this chapter? Write the import statement necessary to use these classes.

2. Write Java code to pop up an option pane asking the user for their age. If the user types a number less than 40, respond with a message box saying that they are young. Otherwise, tease them for being old. For example:



Section 14.2: Graphical Components

3. What is a component? How is a frame different from other components?

4. Name two properties that JFrames have. Give an example piece of code that creates a new JFrame and sets these two properties to have new values of your choice.

5. Identify the Java class used to represent each of the following graphical components.

6. Write a piece of Java code that creates two JButton objects, one with a green background and the text "Click me" and the other with a yellow background and the text "Do not touch!".

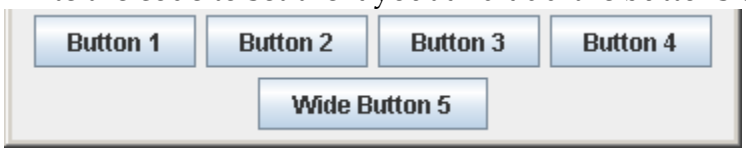
Section 14.3: Laying Out Components in a Frame

For the next three questions, consider the following variable declarations:

```
JButton b1 = new JButton("Button 1");
JButton b2 = new JButton("Button 2");
JButton b3 = new JButton("Button 3");
JButton b4 = new JButton("Button 4");
JButton b5 = new JButton("Button 5");
```

7. Identify the layout manager that would produce the following onscreen appearance. Then write the code to set the layout and add the buttons to the frame appropriately:

8. Identify the layout manager that would produce the following onscreen appearance. Then write the code to set the layout and add the buttons to the frame appropriately:



9. Identify the layout manager that would produce the following onscreen appearance. Then write the code to set the layout and add the buttons to the frame appropriately:

Section 14.4: Events

10. What is an event? What interface is used when handling events?

11. Describe the code that must be written to handle an event. What class(es) and method(s) must be written? What messages must be sent to the component in question (such as the button to be clicked)?

12. Write an `ActionListener` that could be attached to a button, so that whenever that button is clicked, an option pane will pop up that says, `"Greetings, Earthling!"`.

Section 14.5: 2D Graphics

13. What class should be extended when drawing 2D graphics? What method should be overwritten to do the drawing?

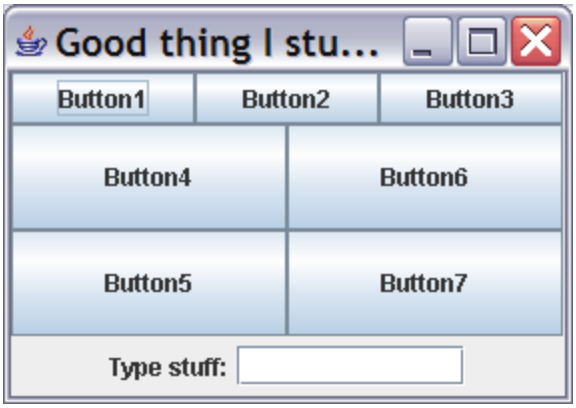
14. Write a panel class that paints a red circle on itself when drawn on the screen.

15. What is a Timer? How are Timers used with panels when drawing 2D graphics?

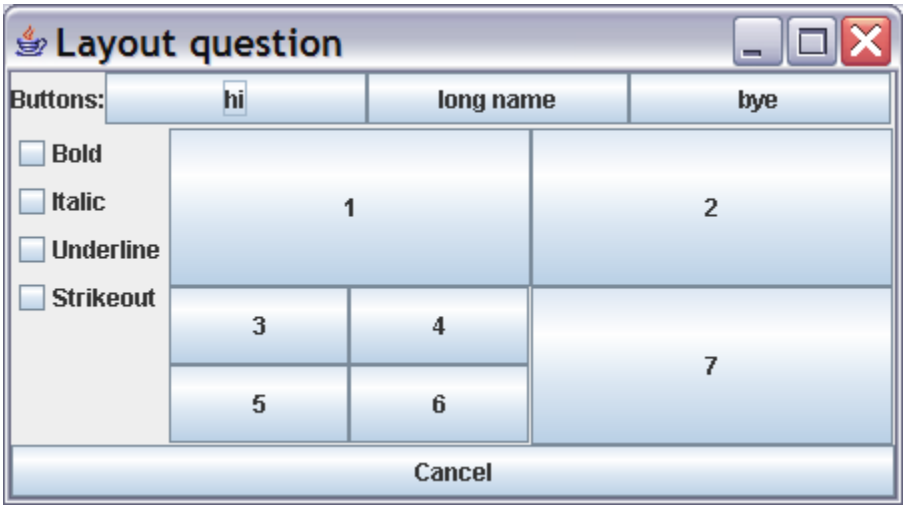
16. Modify your red circle panel from the previous problem to change colors to blue and back again, alternating every 1 second. Use a timer to "animate" the color changes of the panel.

Exercises

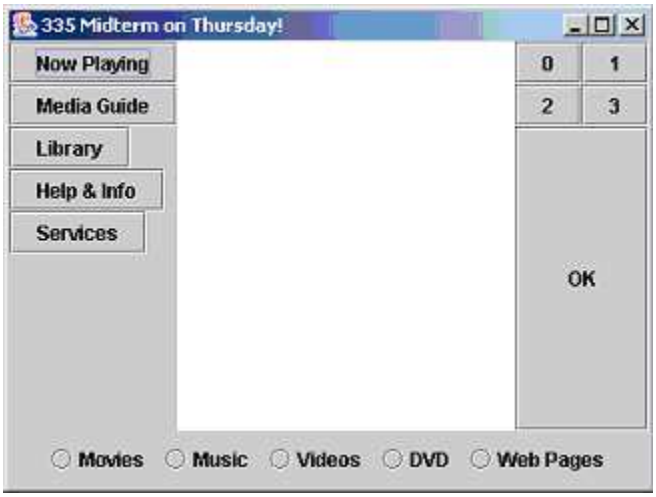
1. Write a complete Java program that creates the following window layout. The window title is `"Good thing I studied!"` and the window size is 285 by 200 pixels.



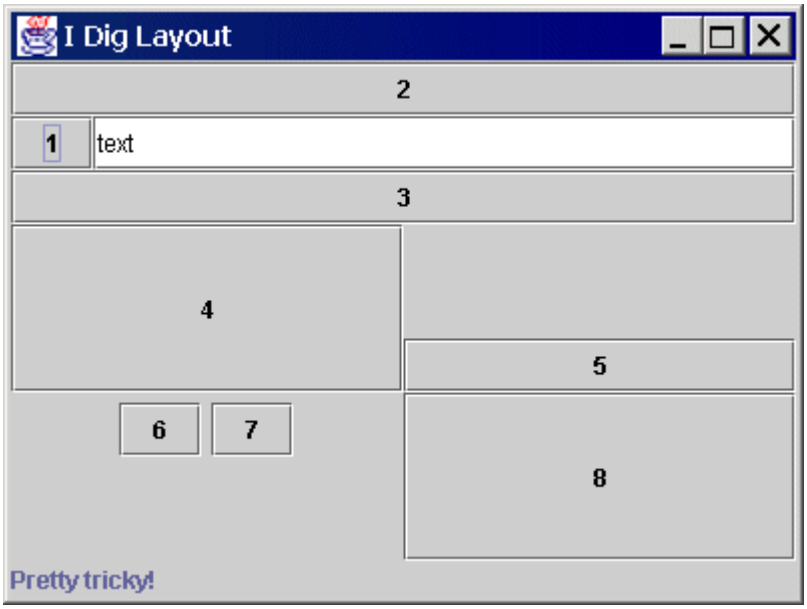
2. Write a complete Java program that creates the following window layout. The window title is `"Good thing I studied!"` and the window size is 420 by 250 pixels.



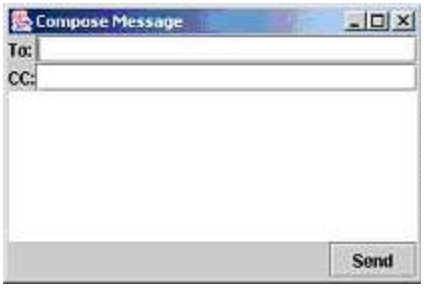
3. Write a complete Java program that creates the following window layout. The window title is `"335 Midterm on Thursday!"` and the window size is 400 by 300 pixels.



4. Write a complete Java program that creates the following window layout. The window title is `"I Dig Layout"` and the window size is 400 by 300 pixels.

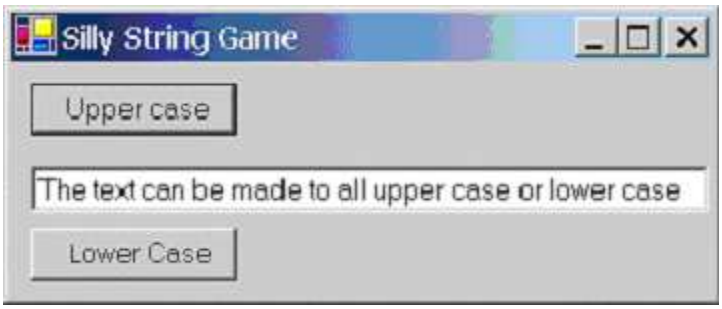


5. Write a complete Java program that creates the following window layout. The window title is `"Compose Message"` and the window size is 300 by 200 pixels.



6. Write a complete Java program that creates the following window. The window title is `"Here's my first Java GUI"` and the window size is packed. Clicking the center button pops up a color chooser to change its background color. Clicking the `"Pick a file..."` button pops up a file chooser.

7. Write a complete Java program that creates the following window layout. The window title is `"Silly String Game"` and the window size is 300 by 100 pixels. The user can type text into the text field, and when the Upper Case button is pressed, the text is capitalized. When the Lower Case button is pressed, the text is made lowercase.



8. Write a complete Java program that creates the following window layout. The window title is `"MegaCalc"` and the window size is 400 by 300 pixels. The window initially should be exactly sized to fit the preferred size of the components inside it. When the user closes the window, the Java program should terminate. In the window there are two text fields, each 4 columns wide, in which the user may type the operands to a binary + operation (in the expression x + y, x and y are the operands and + is the operator). Initially the text in these two fields is blank. Between the operand fields is a button with the text "+".

   Below the previously mentioned fields and button, there is a label which is used to show the result of the adding. Initially, this label's text is a question mark, "?". Below the result label is a button with the text "Clear".

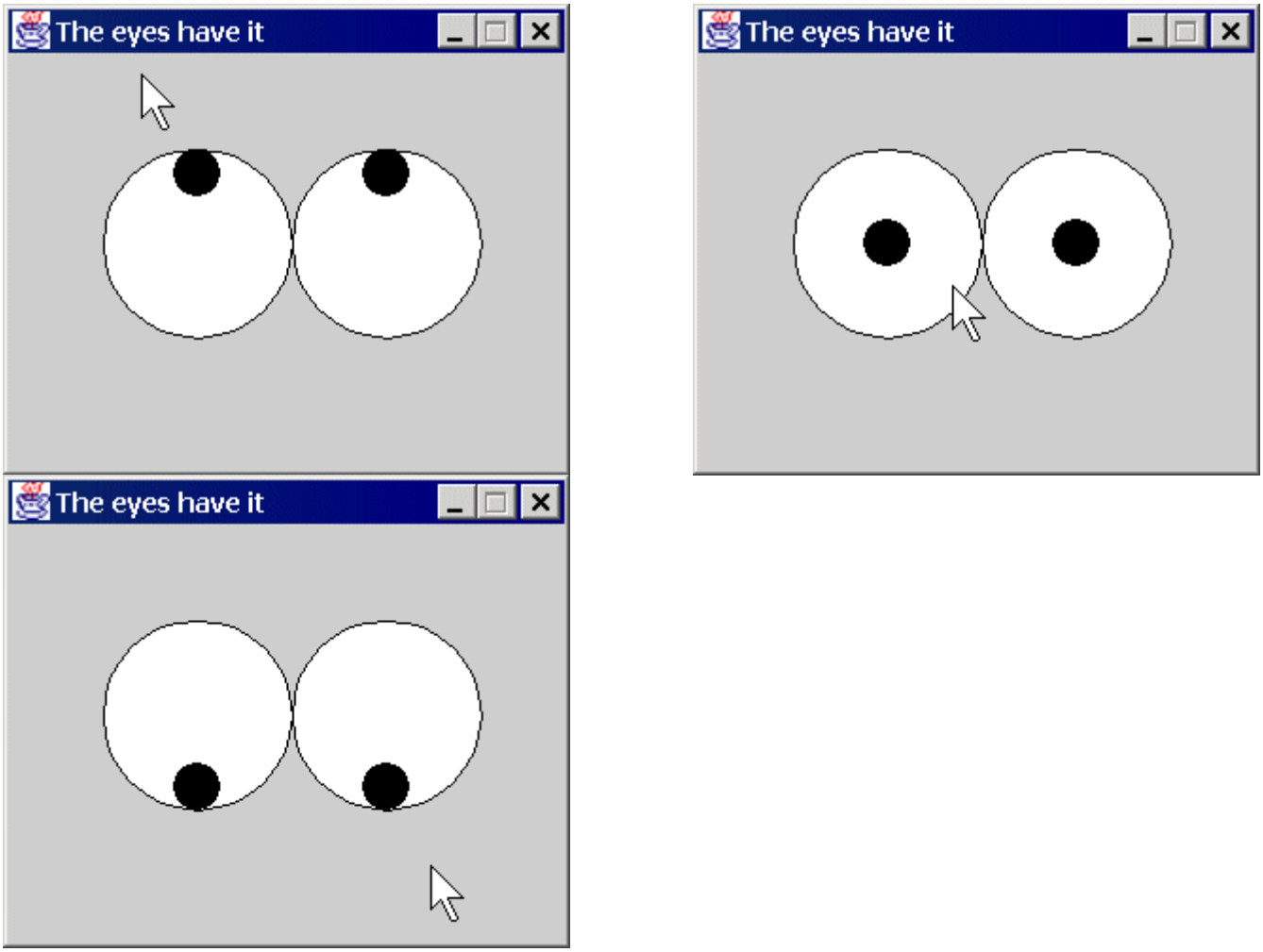
   The event-handling is as follows: When the "+" button is pressed, the text in the two operand fields is treated as two integers; these integers are added, and the resulting integer should be displayed on the screen as the text of the result label. For full credit, you should handle invalid input, such as the possibility that a field is empty or that the user types a non-numeric like "foo" instead of a legal integer, by simply putting "?" as the text for the result field. When the "Clear" button is pressed, the two operand fields are reset to their initial (empty) text, and the result label's text is reset to its initial "?".

9. Write the panel class EyePanel (and any other classes you need, such as event listeners) to implement the drawing and mouse event behavior described below.

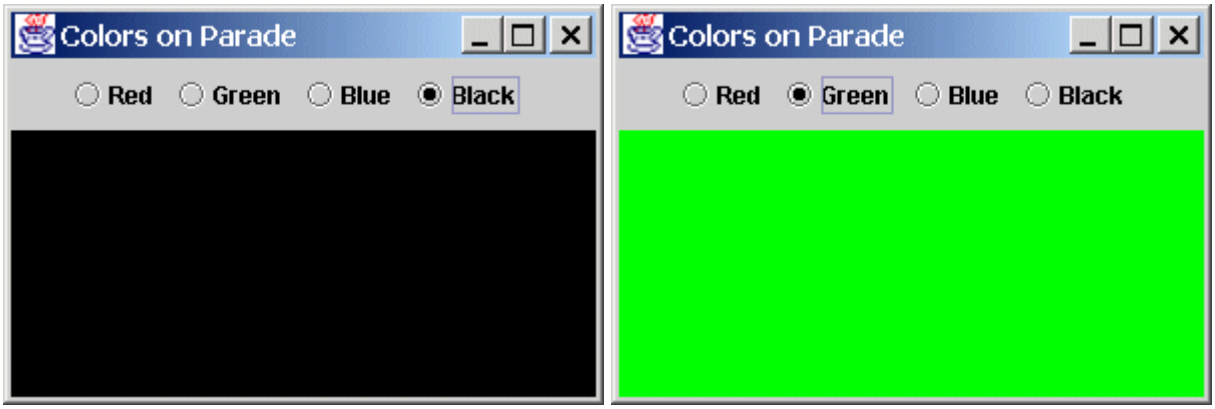
   Appearance: The only visible component in the window frame should be an EyePanel that draws a pair of white circles, with black outlines, to represent eyes. The panel's background color is the default gray. Each eye is 100 x 100 pixels in size. The left eye's top-left corner is at point (50, 50) and the right eye's is at (150, 50). In each eye there is a black circle eyeball of size 25 x 25 pixels. The positions of the eyeballs vary depending on movement of the mouse, as described below.

   Behavior: When the mouse moves in the EyePanel, the panel's appearance should immediately update in the following manner: · If the mouse cursor is above the top of the eyes (its y-coordinate is less than 50), regardless of the x-coordinate, the eyeballs should be drawn "looking up" at coordinates of (87, 50) and (187, 50). · If the mouse cursor is below the bottom of the eyes (its y-coordinate is greater than 150), regardless of the x-coordinate, the eyeballs should be drawn "looking down" at coordinates of (87, 125) and (187, 125). · If the mouse cursor is vertically "inside" the plane of the eyes (y-coordinate between 50 and 150), the eyeballs should be drawn "looking forward" at coordinates of (87, 87) and (137, 87).

   The following three screen shots show the eyes in each of the three positions.

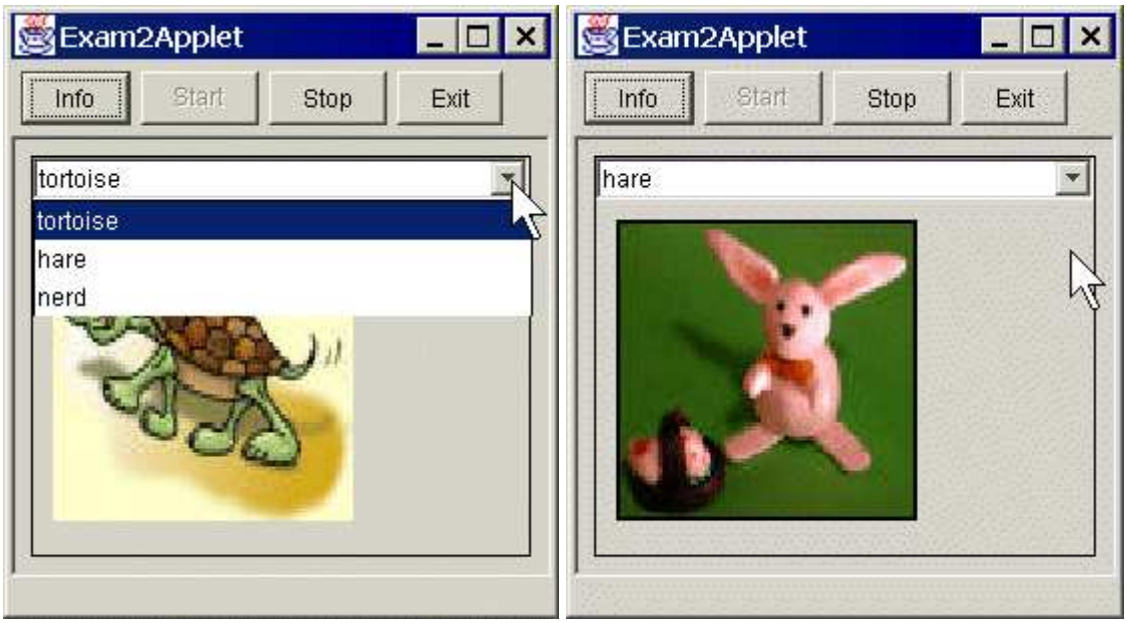


10. Write a complete Java program that uses radio buttons to control a frame's background color, as depicted below.

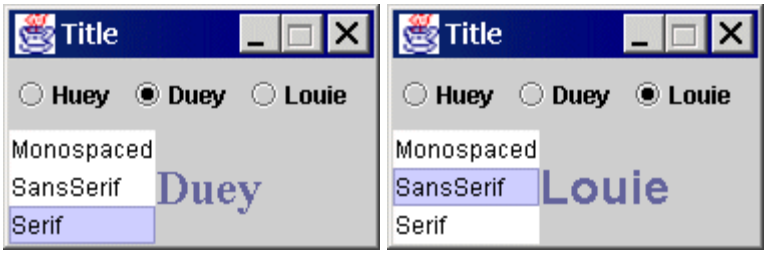


11. Write a complete Java program that uses check boxes and radio buttons to control a label's font style and foreground color, as depicted below.

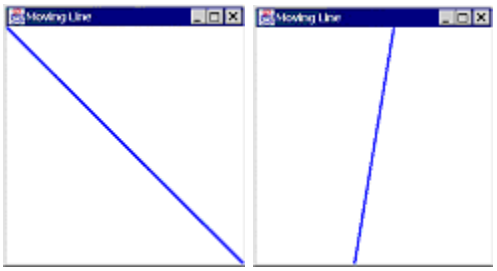
12. Write a complete Java program that uses a combo box to select between several displayable images, as depicted below.



13. Write a complete Java program that uses a list and radio buttons to select between font names and strings to be displayed, as depicted below.

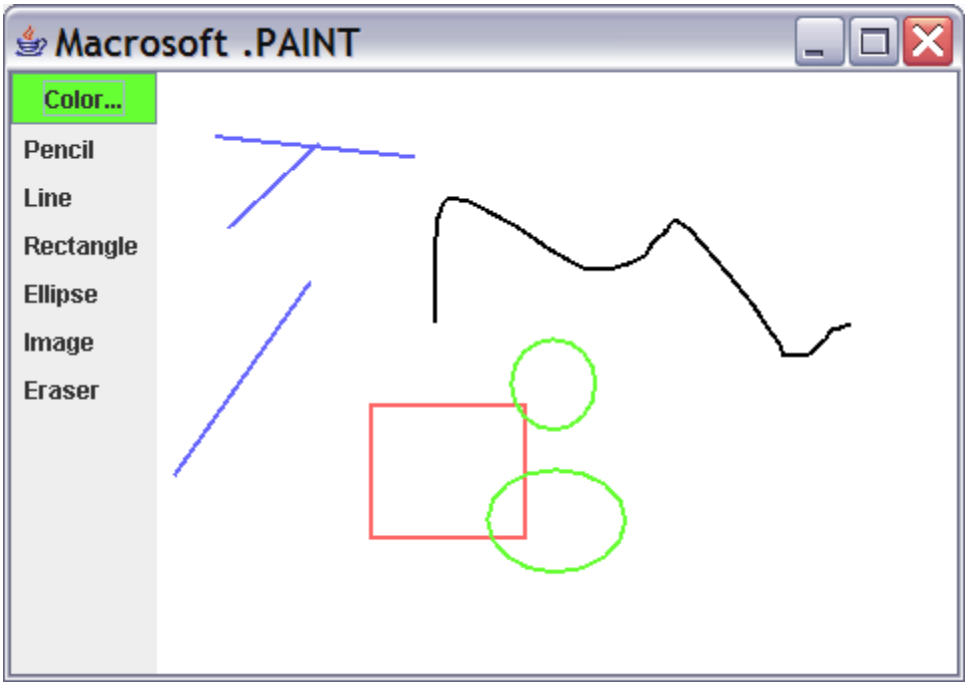


14. Write a complete Java program that animates a moving line, as depicted below.

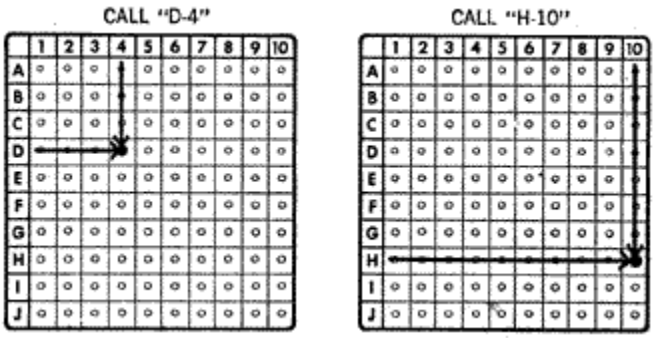


Programming Projects

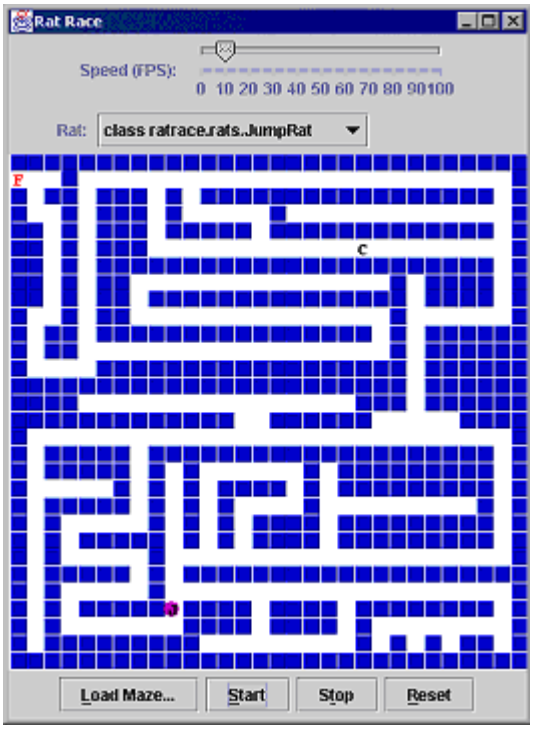
1. Write a painting program that allows the user to paint shapes and lines in many different colors.



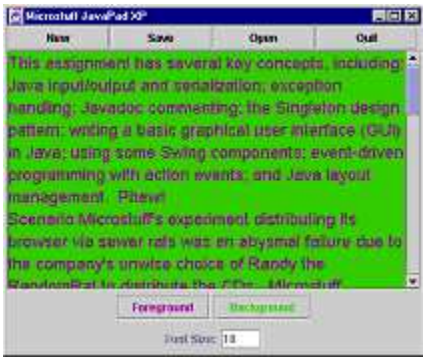
2. Program the game of Battleship.



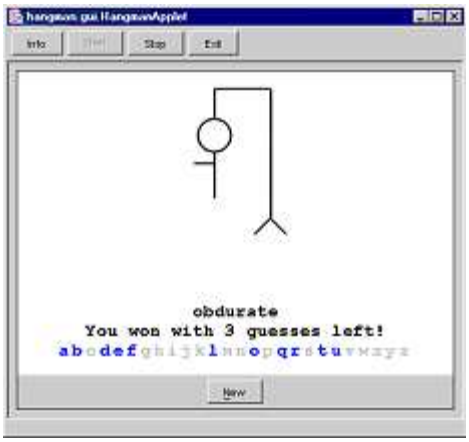
3. Write a program that draws a graphical maze, along with an inheritance hierarchy of rats that escape the maze in different ways. Example: RandomRat moves in a random direction. JumpingRat moves straight but can jump over walls. WallHuggerRat holds on to the right wall until it finds the exit. Consider adding a cat to the maze who can eat any rat it touches.



4. Write a program that represents a simple Notepad clone. Support a New feature (clears the text buffer), a Save feature (writes to a file using a PrintStream), Load (reads from a file using a Scanner), and Quit (exits the program). You can also add support for changing font size and background/foreground colors.



5. Write a graphical Hangman game program. Draw the stick-man with new body parts appearing for each wrong letter guessed. Highlight the letters that have and haven't been guessed yet. Pick the game word from a list you create yourself.



Stuart Reges
Marty Stepp

Appendix A
Answers to Self-Check Problems

Copyright © 2006 by Stuart Reges and Marty Stepp

Chapter 1

1. Computers use binary numbers because it's easier to build electronic devices reliably if they only have to distinguish between two electric states.

2.
   - 6 = 110
   - 44 = 101100
   - 72 = 1001000
   - 131 = 10000011

3.
   - 100 = 4
   - 1011 = 11
   - 101010 = 42
   - 1001110 = 78

4.
   1. Make the cookie batter.
      - Mix the dry ingredients.
      - Cream the butter and sugar.
      - Beat in the eggs.
      - Stir in the dry ingredients.
   2. Bake the cookies.
      - Set the oven for the appropriate temperature.
      - Set the timer.
      - Place the cookies into the oven.
      - Allow the cookies to bake.
   3. Add frosting and sprinkles.
      - Mix the ingredients for the frosting.
      - Spread frosting and sprinkles onto the cookies.

5. MyProgram.java is a source code file typed by the programmer, and MyProgram.class is a compiled executable class file that is run by the computer.

6. The legal identifiers shown are println, AnnualSalary, ABC, sum_of_data, _average, and B4.

7.
```
"Quotes"
Slashes \//
How '"confounding' "\" it is!
```

8.
```
Shaq is 7'1"
The string "" is an empty message.
\'""
```

9.
```
Dear "DoubleSlash" magazine,
    Your publication confuses me.  Is it a
\\ slash or a //// slash that I should use?

Sincerely,
Susan "Suzy" Smith
```

10.
```
System.out.println("\"Several slashes are sometimes seen,\"");
System.out.println("said Sally.  \"I've said so myself.\"  See?");
System.out.println("\\ / \\\\ // \\\\\\ ///");
```

11.
```
System.out.println("This is a test of your");
System.out.println("knowledge of \"quotes\" used");
System.out.println("in 'string literals.'");

System.out.println("You're bound to \"get it right\"");
System.out.println("if you read the section on");
System.out.println("''quotes.''");
```

12.
- The keyword "class" is missing on line 1.
- A semicolon is missing on line 3.
- The word "Println" should not be capitalized on line 4.

13.
- The keyword "void" is missing on line 2.
- The word "string" should be capitalized on line 2.
- A closing " mark is missing on line 4.
- A closing } brace is missing on line 5.

14.
- A { brace is missing on line 1.
- A closing) is missing on line 8.
- The comment on lines 7-9 accidentally comments out lines 8-9 of the program. Using // comments would fix the problem.

15.
```
Inside first method
Inside third method
Inside first method
Inside second method
Inside first method
Inside second method
Inside first method
Inside third method
Inside first method
Inside second method
Inside first method
```

16.
```
Inside first method
Inside first method
Inside second method
Inside first method
Inside third method
Inside second method
Inside first method
Inside first method
Inside second method
Inside first method
Inside third method
```

17.
```
Inside second method
Inside first method
Inside first method
Inside second method
Inside first method
Inside third method
Inside first method
Inside second method
Inside first method
```

18.
```
I am method 1.
I am method 1.
I am method 2.
I am method 3.
I am method 1.
I am method 1.
I am method 2.
I am method 1.
I am method 2.
I am method 3.
I am method 1.
```

19.
```
I am method 1.
I am method 1.
I am method 1.
I am method 2.
I am method 3.
I am method 1.
I am method 2.
I am method 1.
I am method 1.
I am method 2.
I am method 3.
```

20.
```
I am method 1.
I am method 2.
I am method 1.
I am method 1.
I am method 2.
I am method 3.
I am method 1.
I am method 1.
I am method 2.
```

21.
- On line 1, the class name should be LotsOfErrors (no space).
- On line 2, the word 'void' should appear after 'static'.

- On line 2, String should be String[] .
- On line 3, System.println should be System.out.println .
- On line 3, "Hello, world!) should be "Hello, world!") .
- On line 4, there should be a semicolon after message() .
- On line 7, there should be () after message .
- On line 8, System.out println should be System.out.println .
- On line 8, cannot"; should be cannot"); .
- On line 9, the phrase "errors" cannot appear inside a String. 'errors' would work.
- On line 11, there should be a closing } brace.

22.
- Syntax error: The program would not compile because its class name (Demonstration) would not match its file name (Example.java).
- Different program output: The program would not run because Java would be unable to find the main method.
- Different program output: There would now be a blank line between the two printed messages.
- Syntax error: The program would not compile because the main method would be calling a method displayRule that no longer existed.
- No effect. The program would still compile successfully and produce the same output.
- Different program output: The output would now have no line break between "The first rule " and "of Java Club is," in its output.

23.
```
1  public class GiveAdvice {
2      public static void main(String[] args) {
3          System.out.println("Programs can be easy or difficult");
4          System.out.println("to read, depending upon their format.");
5          System.out.println();
6          System.out.println("Everyone, including yourself, will be");
7          System.out.println("happier if you choose to format your");
8          System.out.println("Programs.");
9      }
10 }
```

24.
```
1  public class Messy {
2      public static void main(String[] args) {
3          message();
4          System.out.println();
5          message();
6      }
7
8      public static void message() {
9          System.out.println("I really wish that");
10         System.out.println("I had formatted my source");
11         System.out.println("code correctly!");
12     }
13 }
```

Chapter 2

1. 22, -1, and -6875309 are legal int literals.

2.

- 8
- 11
- 6
- 4
- 33
- -16
- 6.4
- 6
- 30
- 1
- 7
- 5
- 2
- 18
- 3
- 4
- 4
- 15
- 8

3.

- 1
- 9.0
- 9.6
- 2.2
- 6.0
- 6.0
- 2.2
- 8.0
- 1.25
- 3.0
- 3.0
- 3.0
- 5.0
- 6.4
- 37.0
- 9.0
- 8.5
- 9.6
- 4.0
- 4.8

4.

- 11
- "2 + 2 34"
- "2 2 + 3 4"
- "7 2 + 2"
- "2 + 2 7"
- "(2 + 2) 7"
- "hello 34 8"

5.
```
int age;
String gender;
double height;
int weight;
```

6.
```
String year;
int numberOfCourses;
double gpa;
```

7. Last digit: `number % 10`

8. Second-to-last digit: `(number % 100) / 10` or `(number / 10) % 10`
   Third-to-last digit: `(number % 1000) / 100` or `(number / 100) % 10`

9.
```
first: 19
second: 8
```

   The code swaps the values of the variables first and second.

10.
```
int first = 8, second = 19;
first += second;
second = first - second;
first -= second;
```

11.
```
a: 7
b: 10
c: 16
```

12.
- 15 * count - 11
- -10 * count + 40
- 4 * count - 11
- -3 * count + 100

13.
```
for (int i = 1; i < 6; i++) {
    // your code here
    System.out.println(18 * i - 22);
}
```

14.
```
System.out.println("Twas brillig and the ");
System.out.println("   slithy toves did gyre and");
System.out.println("gimble");
System.out.println();
System.out.println("in the wabe.");
```

15.
- The loop prints every third number, not every odd number. The statement `count = count + 2` on line 8 should be moved into the loop header instead of `count++`.
- On line 12, the variable count is no longer defined (its scope is only within the for loop above). It should be declared before the loop begins rather than inside the loop's header.
- On line 13, too large a value is printed for the final odd number; `count` should be printed, not count + 2.
- On line 20, it is illegal to try to assign a new value to a constant such as MAX_ODD. One way to fix this would be to write two methods: one to print the odds up to 21 and a second to print the odds up to 11. But this is admittedly redundant. The better solution to this kind of problem is called parameter passing, which will be seen in later chapters.

16.
```
4
2
```

17.
```
The result is: 55
```

18.
```
24  1
22  2
19  3
15  4
10  5
```

19.
```
+-----+
\     /
/     \
\     /
/     \
\     /
/     \
+-----+
```

20.
```
How many lines
How many lines
How many lines
are printed?
```

21.
```
T-minus 5, 4, 3, 2, 1, Blastoff!
```

22.
```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
```

23.
```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
 *******************
```

24.
```
****!****!****!
****!****!****!
```

25.
```
************!
************!
```

26.
```
*!*!*!*!
*!*!*!*!
*!*!*!*!
*!*!*!*!
*!*!*!*!
*!*!*!*!
```

27.
```
1  public class SlashFigure {
2      public static void main(String[] args) {
3          for (int line = 1; line <= 6; line++) {
4              for (int i = 1; i <= 2 * line - 2; i++) {
5                  System.out.print("\\");
6              }
7              for (int i = 1; i <= -4 * line + 26; i++) {
8                  System.out.print("!");
9              }
10             for (int i = 1; i <= 2 * line - 2; i++) {
11                 System.out.print("/");
12             }
13             System.out.println();
14         }
15     }
16 }
```

28.
```
1  public class SlashFigure2 {
2      public static final int SIZE = 4;
3
4      public static void main(String[] args) {
5          for (int line = 1; line <= SIZE; line++) {
6              for (int i = 1; i <= 2 * line - 2; i++) {
7                  System.out.print("\\");
8              }
9              for (int i = 1; i <= -4 * line + (4 * SIZE + 2); i++) {
10                 System.out.print("!");
11             }
12             for (int i = 1; i <= 2 * line - 2; i++) {
13                 System.out.print("/");
14             }
15             System.out.println();
16         }
17     }
18 }
```

Chapter 3

1.
```
1 3 5
1 3 5 7 9 11 13 15
1 3 5 7 9 11 13 15 17 19 21 23 25
```

2.
```
1 2 3 4 5
1 2 3 4 5 6 7
1 2 3 4
number = 8
```

3.
```
three times two = 6
1 times three = 28
1 times 1 = 42
three times 1 = 2
1 times eight = 20
```

4. ```
   whom and who like it
   it and him like whom
   whom and him like him
   stu and boo like who
   her and him like who
   ```

5. ```
   public static void printStrings(String s, int n) {
       for (int i = 1; i <= n; i++) {
           System.out.print(s);
       }
       System.out.println();
   }
   ```

6. System.out.println is an overloaded method.

7. The program changes the value of its parameter tempc, but this doesn't affect the variable tempc in main. The incorrect output is:

   ```
   Body temp in C is: 0.0
   ```

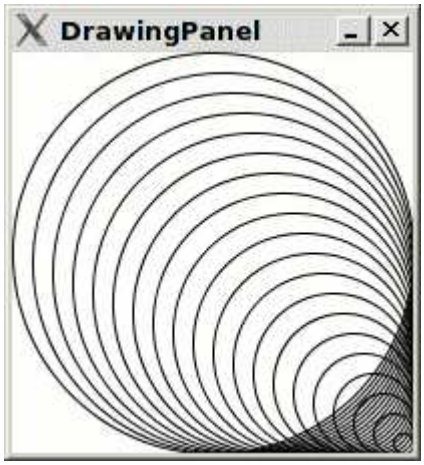
8. - 1.6
   - 2
   - 36.0
   - 64.0
   - 10.0
   - 116.0
   - 7
   - 5
   - -5
   - 8.0
   - 11.0
   - 102.0
   - 17.0
   - 20.0
   - 13.0
   - 14.0

9. ```
   3 0
   1 2 4
   4 3
   5 2 4
   8 1
   5 9 4
   ```

10. ```
    public static int min(int n1, int n2, int n3) {
        return Math.min(n1, Math.min(n2, n3));
    }
    ```

11. ```
    public static int countQuarters(int cents) {
        return cents % 100 / 25;
    }
    ```

12. - 6
    ```

- 19
- q.e.d.
- ARCTURAN MEGADONKEY
- The code throws an IndexOutOfBoundsException.
- egad
- 4
- 1
- 13
- -1
- Arcturan Megadonkeys
- b
- Cyber
- mega Corp

13.
```
quote.substring(5, 10).toUpperCase()
quote.toLowerCase().substring(0, 4) + quote.substring(20, 26)
```

14.
```
p1: java.awt.Point[x=20,y=10]
p2: java.awt.Point[x=46,y=4]
p3: java.awt.Point[x=46,y=4]
```

15.
```
(5, 2)
(6, -3)
(5, 2)
```

16. There are 10 tokens:
    - Hello : String
    - there. : String
    - 1+2 : String
    - is : String
    - 3 : int, double, String
    - and : String
    - 1.5 : double, String
    - squared : String
    - is : String
    - 2.25! : String

17.
    - 34.50 : The code will run successfully and the variable money will store the value 34.5.
    - 6 : The code will run successfully and the variable money will store the value 6.0.
    - $25.00 : The code will crash with an exception.
    - million : The code will crash with an exception.
    - 100*5 : The code will crash with an exception.
    - 600,000 : The code will crash with an exception.
    - none : The code will crash with an exception.
    - 645. : The code will run successfully and the variable money will store the value 645.0.

18.
```
Scanner console = new Scanner(System.in);
System.out.print("Type an integer: ");
int number = console.nextInt();
System.out.println(number + " times 2 = " + (number * 2));
```

19.
```
Scanner console = new Scanner(System.in);
System.out.print("What is your phrase? ");
String phrase = console.nextLine();
System.out.print("How many times should I repeat the phrase? ");
int times = console.nextInt();

for (int i = 1; i <= times; i++) {
    System.out.println(phrase);
}
```

Supplement 3G

1. 1. On the second line, the call to drawLine should be made on the Graphics object in the panel, not on the DrawingPanel itself.
   2. On the second line, the order of the parameters is incorrect. They should be 50, 86, 20, and 35. The following is the corrected code:

      ```
      DrawingPanel panel = new DrawingPanel(200, 200);
      Graphics g = panel.getGraphics();
      panel.drawLine(50, 86, 20, 35);
      ```

2. The black rectangle is being drawn second, so it's covering up the white inner circle. The following code fixes the problem:

   ```
   DrawingPanel panel = new DrawingPanel(200, 100);
   Graphics g = panel.getGraphics();
   g.setColor(Color.BLACK);
   g.fillRect(10, 10, 50, 50);
   g.setColor(Color.WHITE);
   g.fillOval(10, 10, 50, 50);
   ```

3. The problem is that the parameters for the drawRect and drawLine methods have different meanings. In drawRect, the parameters are (x, y, width, height), and in drawLine, they are (x1, y1, x2, y2). To fix the problem, the third and fourth parameters passed to drawRect should be changed to 40 and 20 so that the rectangle's bottom left corner will be at (50, 40). The following code fixes the problem:

   ```
   DrawingPanel panel = new DrawingPanel(200, 100);
   Graphics g = panel.getGraphics();
   g.drawRect(10, 20, 40, 20);
   g.drawLine(10, 20, 50, 40);
   ```

4. The Draw7 program draws black circles that get smaller and smaller, each circle with its right and bottom edges touching the right and bottom corners of the window.



Chapter 4

1. The sum variable needs to be declared outside the for loop. The following code fixes the problem:

```
public static int sumTo(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

2. The code has a fencepost problem; a comma will be printed after the last number. The following code fixes the problem.

```
System.out.print(1);
for (int i = 1; i <= n; i++) {
    System.out.print(", " + i);
}
System.out.println();  // end the line of output
```

3.
```
Scanner console = new Scanner(System.in);
System.out.print("How many numbers? ");
int count = console.nextInt();
int product = 1;

for (int i = 1; i <= count; i++) {
    System.out.print("Next number --> ");
    int num = console.nextInt();
    product *= num;
}

System.out.println("Product = " + product);
```

4.
   - `z % 2 == 1`
   - `z <= Math.sqrt(y)`
   - `y > 0`
   - `x % 2 != y % 2`
   - `y % z == 0`
   - `z != 0`

- `Math.abs(y) > Math.abs(z)`
- `(x >= 0) == (z < 0)`
- `y % 10 == y`
- `z >= 0`
- `x % 2 == 0`
- `Math.abs(x - y) < Math.abs(z - y)`

5.
- `true`
- `false`
- `true`
- `false`
- `true`
- `false`
- `false`
- `true`
- `true`

6. The code incorrectly uses an if/else/if/else/if pattern, when it should really use an if/if/if pattern, because the three conditions are not mutually exclusive of each other. The given code only answers correctly when zero or one of the numbers are odd; if more than one is odd, the code only enters one branch and does not properly increment the counter multiple times.

   The following version of the code fixes the problem:

```
if (n1 % 2 == 1) {
    count++;
}
if (n2 % 2 == 1) {
    count++;
}
if (n3 % 2 == 1) {
    count++;
}
```

   Or, the following version achieves the same thing without the need for if/else statements:

```
count = n1 % 2 + n2 % 2 + n3 % 2;
```

7.
```
Scanner console = new Scanner(System.in);

System.out.print("Type a number: ");
int number = console.nextInt();

if (number % 2 == 0) {
    System.out.println("even");
} else {
    System.out.println("odd");
}
```

8. The code incorrectly prints that even numbers not divisible by 3 are odd. This is because the else statement matches the most closely nested if statement (number % 3 == 0), not the outer if statement.

The following change corrects the problem. Note the braces around the outer if statement.

```
if (number % 2 == 0) {
    if (number % 3 == 0) {
        System.out.println("Divisible by 6.");
    }
}
else {
    System.out.println("Odd.");
}
```

9. The code shouldn't return the factor i when found; it should instead count how many factors it finds. This is a cumulative sum, so the count variable should be declared outside the for loop. The following code fixes the problem:

```
public static int countFactors(int n) {
    int count = 0;
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {  // factor
            count++;
        }
    }
    return count;
}
```

10.
```
public static void moneyMultiply(int sum, int total, int count1, int count2) {
    Scanner console = new Scanner(System.in);
    System.out.print("Is your money multiplied 1 or 2 times? ");
    int times = console.nextInt();

    System.out.print("And how much are you contributing? ");
    int donation = console.nextInt();
    sum += times * donation;
    total += donation;

    if (times == 1) {
        count1++;
    }
    else if (times == 2) {
        count2++;
    }
}
```

If the user could type any number, the code might need additional if statements to increment the proper count variable. If the user could type anything, even a non-integer, the code might need to use the `hasNextInt` method of the Scanner to ensure valid input before proceeding.

11.
```
1  public class Bills {
2      public static void main(String[] args) {
3          Scanner console = new Scanner(System.in);
4
5          int numBills1 = getBills(console, "John");
6          int numBills2 = getBills(console, "Jane");
7
8          System.out.println("John needs " + numBills1 + " bills");
9          System.out.println("Jane needs " + numBills2 + " bills");
10     }
11
12     public static int getBills(Scanner console, String name) {
13         System.out.print("How much will " + name + " be spending? ");
14         double amount = console.nextDouble();
15         System.out.println();
16
17         int numBills = (int) (amount/20.0);
18         if (numBills * 20.0 < amount) {
19             numBills++;
20         }
21
22         return numBills;
23     }
24 }
```

12. The code won't ever print "Mine, too!" because Strings cannot be compared with the == operator. The fourth line of the code should be changed to the following:

```
if (name.equals("blue")) {
```

13. The code's output is the following, because the == operator was used instead of the equals method:

```
non-equal
```

14.
```
second
third
```

15.
```
Scanner console = new Scanner(System.in);
System.out.print("What color do you want? ");
String choice = console.nextLine();
if (choice.equalsIgnoreCase("r")) {
    System.out.println("You have chosen Red.");
} else if (choice.equalsIgnoreCase("g")) {
    System.out.println("You have chosen Green.");
} else if (choice.equalsIgnoreCase("b")) {
    System.out.println("You have chosen Blue.");
} else {
    System.out.println("Unknown color: " + choice);
}
```

16.
```
Scanner console = new Scanner(System.in);
System.out.print("Enter a card: ");
String rank = console.next();
String suit = console.next();

if (rank.equals("2")) {
    rank = "Two";
} else if (rank.equals("3")) {
    rank = "Three";
} else if (rank.equals("4")) {
    rank = "Four";
} else if (rank.equals("5")) {
    rank = "Five";
} else if (rank.equals("6")) {
    rank = "Six";
} else if (rank.equals("7")) {
    rank = "Seven";
} else if (rank.equals("8")) {
    rank = "Eight";
} else if (rank.equals("9")) {
    rank = "Nine";
} else if (rank.equals("10")) {
    rank = "Ten";
} else if (rank.equals("J")) {
    rank = "Jack";
} else if (rank.equals("Q")) {
    rank = "Queen";
} else if (rank.equals("K")) {
    rank = "King";
} else {  // rank.equals("A")
    rank = "Ace";
}

if (suit.equals("C")) {
    suit = "Clubs";
} else if (suit.equals("D")) {
    suit = "Diamonds";
} else if (suit.equals("H")) {
    suit = "Hearts";
} else {  // suit.equals("S")
    suit = "Spades";
}

System.out.println(rank + " of " + suit);
```

17. The expression equals 6.800000000000001 because of the limited precision of the double type led to a roundoff.

18. The expression gpa * 3 equals 9.600000000000001 because of double roundoff. A fix would be to test that the value was close to 9.6 rather than exactly equal to it.

```
double gpa = 3.2;
if (Math.abs(gpa * 3 - 9.6) < 0.1) {
    System.out.println("You earned enough credits.");
}
```

19.
```
if (Character.isUpperCase(theString.charAt(0))) {
    ...
}
```

20. The toLowerCase method cannot be called on a char value, which is what the charAt method returns. A better solution would be to call the Character.toLowerCase method on the characters of the string.

```
int count = 0;
for (int i = 0; i < s.length(); i++) {
    if (Character.toLowerCase(s.charAt(i)) == 'e') {
        count++;
    }
}
```

Another solution would be to lowercase the entire string once before the loop.

```
s = s.toLowerCase();
int count = 0;
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == 'e') {
        count++;
    }
}
```

21.
```
String name = "Marla Singer";
int space = name.indexOf(" ");
String lastName = name.substring(space + 1);
String firstInitial = name.substring(0, 1);
String lastNameFirstInitial = lastName + ", " + firstInitial + ".";
System.out.println(lastNameFirstInitial);
```

or, a shorter version:

```
String name = "Marla Singer";
System.out.println(name.substring(name.indexOf(" ") + 1) +
        ", " + name.charAt(0) + ".");
```

22.
```
// assuming that the string is stored in variable 'str'
int count = 0;
for (int i = 0; i < str.length(); i++) {
    if (Character.toLowerCase(str.charAt(i)) >= 'n') {
        count++;
    }
}
System.out.println(count + " letters come after n.");
```

23.
```
public static void printTriangleType(int s1, int s2, int s3) {
    if (s1 == s2) {
        if (s2 == s3) {
            System.out.println("equilateral");
        }
    } else if (s2 == s3) {
        System.out.println("isosceles");
    } else {
        System.out.println("scalene");
    }
}
```

Invalid values are when a side's length is negative, or when any one side length is greater than the sum of the other two side lengths, because this cannot be a valid triangle. The precondition of of the printTriangleType method is that the side lengths constitute a valid triangle.

24. The preconditions of this method are that the grade parameter's value is between 0 and 100.

25. The code fails when n3 is the smallest of the three numbers; for example, when the parameters' values are (4, 7, 2), the code should return 4 but instead returns 2. The method could be correctly written as:

```
public static int medianOf3(int n1, int n2, int n3) {
    if (n1 < n2 && n1 < n3) {
        if (n2 < n3) {
            return n2;
        } else {
            return n3;
        }
    } else if (n2 < n1 && n2 < n3) {
        if (n1 < n3) {
            return n1;
        } else {
            return n3;
        }
    } else { // (n3 < n1 && n3 < n2)
        if (n1 < n2) {
            return n1;
        } else {
            return n2;
        }
    }
}
```

or the following shorter version:

```
public static int medianOf3_2(int n1, int n2, int n3) {
    return Math.max(Math.max(Math.min(n1, n2), Math.min(n2, n3)), Math.min(n1, n3));
}
```

26. Invalid values are when $a = 0$ (because it makes the denominator of the equation equal 0), or when $b^2 - 4ac < 0$, because then it has no real square root.

```
// Throws an exception if a, b, c are invalid.
public static void quadratic(int a, int b, int c) {
    double determinant = b * b - 4 * a * c;

    if (a == 0) {
        throw new IllegalArgumentException("Invalid a value of 0");
    }
    if (determinant < 0) {
        throw new IllegalArgumentException("Invalid determinant");
    }

    ...
}
```

Chapter 5

1.

• executes body 10 times • `1 11 21 31 41 51 61 71 81 91`	• executes body 3 times • `2 4 16`
• executes body 0 times • (no output)	• executes body 5 times • `bbbbbabbbbb`

• loops infinitely
• `250`
`250`
`250`
.... (repeats forever)
	• executes body 7 times
	• `10`
	`5`
	`2`
	`1`
	`0`
	`0`
	`0`

2.

- ```
  int n = 1;
  while (n <= max) {
      System.out.println(n);
      n++;
  }
  ```

- ```
  int total = 25;
  int number = 1;
  while (number <= (total / 2)) {
      total = total - number;
      System.out.println(total + "  " + number);
      number++;
  }
  ```

- ```
  int i = 1;
  while (i <= 2) {
      int j = 1;
      while (j <= 3) {
          int k = 1;
          while (k <= 4) {
              System.out.print("*");
              k++;
          }
          System.out.print("!");
          j++;
      }
      System.out.println();
      i++;
  }
  ```

- ```
  int number = 4;
  int count = 1;
  while (count <= number) {
      System.out.println(number);
      number = number / 2;
      count++;
  }
  ```

3.
```
Method Call                          Output
--------------------------------------
mystery(1);                          1 0
mystery(6);                          4 2
mystery(19);                         16 4
mystery(39);                         32 5
mystery(74);                         64 6
```

4.
```
Method Call                          Output
--------------------------------------
mystery(19);                         19 0
mystery(42);                         21 1
mystery(48);                         3 4
mystery(40);                         5 3
mystery(64);                         1 6
```

5.
```
int SENTINEL = -1;

System.out.print("Type a number (or " + SENTINEL + " to stop): ");
Scanner console = new Scanner(System.in);
int input = console.nextInt();
int min = input;
int max = input;

while (input != SENTINEL) {
    if (input < min) {
        min = input;
    } else if (input > max) {
        max = input;
    }

    System.out.print("Type a number (or " + SENTINEL + " to stop): ");
    input = console.nextInt();
}

if (min != SENTINEL) {
    System.out.println("Maximum was " + max);
    System.out.println("Minimum was " + min);
}
```

6.
```
public static int zeroDigits(int number) {
    int count = 0;
    do {
        if (number % 10 == 0) {
            count++;
        }
        number = number / 10;
    } while (number > 0);
    return count;
}
```

7.
```
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int number = console.nextInt();
String textNumber = String.valueOf(number);

// print odd digits
for (int i = 0; i < textNumber.length(); i++) {
    int digit = Character.getNumericValue(textNumber.charAt(i));
    if (digit % 2 == 1) {
        System.out.print(digit);
    }
}

// print even digits
for (int i = 0; i < textNumber.length(); i++) {
    int digit = Character.getNumericValue(textNumber.charAt(i));
    if (digit % 2 == 0) {
        System.out.print(digit);
    }
}
```

8. a: 0 through 99 inclusive
   b: 50 through 69 inclusive
   c: 0 through 69 inclusive
   d: -20 through 79 inclusive
   e: 0, 4, 8, 16, 20, 24, 28, 32, or 36

9. ```
   Random rand = new Random();
   int num = rand.nextInt(11);
   ```

10. ```
    Random rand = new Random();
    int num = rand.nextInt(25) * 2 + 51;
    ```

11. - true
    - true
    - false
    - true
    - true
    - false
    - false
    - true
    - true
    - true
    - true
    - false

12. ```
    public static boolean isVowel(char c) {
        c = Character.toLowerCase(c);  // case-insensitive
        return c == 'a' || c == 'e' || c == 'i' ||
               c == 'o' || c == 'u';
    }
    ```

    or,

    ```
    public static boolean isVowel2(char c) {
        String vowels = "aeiouAEIOU";
        return vowels.indexOf(c) >= 0;
    }
    ```

13. In this code the boolean flag isn't being used properly, because if the code finds a factor of the number, prime will be set to false, but on the next pass through the loop, if the next number isn't a factor, prime will be reset back to true again. The following code fixes the problem:

    ```
    public static boolean isPrime(int n) {
        boolean prime = true;
        for (int i = 2; i < n; i++) {
            if (n % i == 0) {
                prime = false;
            }
        }

        return prime;
    }
    ```
    ```

14. In this code the boolean flag isn't being used properly, because if the code finds the character, found will be set to true, but on the next pass through the loop, if the next character isn't ch, found will be reset back to false again. The following code fixes the problem:

```
public static boolean contains(String str, char ch) {
    boolean found = false;
    for (int i = 0; i < str.length(); i++) {
        if (str.charAt(i) == ch) {
            found = true;
        }
    }

    return found;
}
```

15.
```
public static boolean startEndSame(String str) {
    return str.charAt(0) == str.charAt(str.length() - 1);
}
```

16.
```
public static boolean hasPennies(int cents) {
    return cents % 5 != 0;
}
```

17.
```
Method Call                     Value Returned
-----------------------------------------------
mystery(3, 3)                    3
mystery(5, 3)                    1
mystery(2, 6)                    2
mystery(12, 18)                  6
mystery(30, 75)                  15
```

18. The code should re-prompt for a valid integer for the user's age and a valid real number for the user's GPA. If the user types a token of the wrong type, their line of input should be consumed, and they should be reprompted.

    The following code implements the corrected behavior:

```
Scanner console = new Scanner(System.in);
System.out.print("Type your age: ");
while (!console.hasNextInt()) {
    console.nextLine();  // throw away the offending token
    System.out.print("Type your age: ");
}
int age = console.nextInt();

System.out.print("Type your GPA: ");
while (!console.hasNextDouble()) {
    console.nextLine();  // throw away the offending token
    System.out.print("Type your GPA: ");
}
double gpa = console.nextDouble();
```

19.
    - When the user types `Jane`:

```
Type something for me! Jane
Your name is Jane
```

- When the user types `56`:

```
Type something for me! 56
Your IQ is 56
```

- When the user types `56.2`:

```
Type something for me! 56.2
Your name is 56.2
```

20.
```
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
if (console.hasNextDouble()) {
    double value = console.nextDouble();
    System.out.println("You typed the real number " + value);
}
else if (console.hasNextInt()) {
    int value = console.nextInt();
    System.out.println("You typed the integer " + value);
}
```

21.
```
String prompt = "Please enter a number: ";
Scanner console = new Scanner(System.in);

int num1 = getInt(console, prompt);
int num2 = getInt(console, prompt);
int num3 = getInt(console, prompt);

double average = (num1 + num2 + num3) / 3.0;
System.out.println("Average: " + average);
```

22.

• executes body 10 times • `1 11 21 31 41 51 61 71 81 91`	• executes body 3 times • `2 4 16`
• infinite • `count down: 10` `count down: 9` .... (repeats forever)	• executes body 5 times • `bbbbbabbbbb`

• loops infinitely
• `250`
`250`
`250`
.... (repeats forever)
	• executes body 7 times
	• `10`
	`5`
	`2`
	`1`
	`0`
	`0`
	`0`

• executes body 2 times • `100` `50`	• executes body 3 times • `/\/\/\/\/\/\/\/\`

23.
```
Scanner console = new Scanner(System.in);
String response;
do {
    System.out.println("She sells seashells by the seashore.");
    System.out.print("Do you want to hear it again? ");
    response = console.nextLine();
} while (response.equals("y"));
```

24.
```
Scanner console = new Scanner(System.in);
Random rand = new Random();

int num;
do {
    num = rand.nextInt(1000);
    System.out.println("Random number: " + num);
} while (num < 900);
```

25.
```
int SENTINEL = -1;

Scanner console = new Scanner(System.in);
int input = console.nextInt();
int min = input;
int max = input;

while (true) {
    System.out.print("Type a number (or " + SENTINEL + " to stop): ");
    input = console.nextInt();
    if (input == SENTINEL) {
        break;
    }

    if (input < min)
        min = input;
    else if (input > max)
        max = input;
}

if (min != SENTINEL || max != SENTINEL) {
    System.out.println("Maximum was " + max);
    System.out.println("Minimum was " + min);
}
```

26.

	y < x	y == 0	count > 0
Point A:	SOMETIMES	SOMETIMES	NEVER
Point B:	ALWAYS	SOMETIMES	SOMETIMES
Point C:	ALWAYS	ALWAYS	ALWAYS
Point D:	SOMETIMES	SOMETIMES	SOMETIMES
Point E:	NEVER	SOMETIMES	SOMETIMES

27.

	n > b	a > 1	b > a
Point A:	SOMETIMES	SOMETIMES	SOMETIMES
Point B:	ALWAYS	SOMETIMES	SOMETIMES
Point C:	SOMETIMES	ALWAYS	ALWAYS
Point D:	SOMETIMES	ALWAYS	NEVER
Point E:	NEVER	SOMETIMES	SOMETIMES

28.
```
          next == 0   prev == 0   next == prev
Point A: SOMETIMES   ALWAYS      SOMETIMES
Point B: NEVER       SOMETIMES   SOMETIMES
Point C: NEVER       NEVER       ALWAYS
Point D: SOMETIMES   NEVER       SOMETIMES
Point E: ALWAYS      SOMETIMES   SOMETIMES
```

Chapter 6

1. `Scanner input = new Scanner(new File("input.txt"));`

2. The Scanner should read a new File with the name "test.dat". The correct line of code is:

   ```
   Scanner input = new Scanner(new File("test.dat"));
   ```

3. The file name String should use / or \\ instead of \. The \ is used to create escape sequences and \\ represents a literal backslash. The correct String is:

   ```
   Scanner input = new Scanner(new File("C:/temp/new files/test.dat"));
   ```

4. `Scanner input = new Scanner(new File("input.txt"));`

5. - `"numbers.dat"` or `"C:/Documents and Settings/amanda/My Documents/programs/numbers.dat"`
   - `"data/homework6/input.dat"` or `"C:/Documents and Settings/amanda/My Documents/programs/data/homework6/input.dat"`
   - There is only one legal way to refer to this file: by its absolute path, `"C:/Documents and Settings/amanda/My Documents/homework/data.txt"`

6. - `"names.txt"` or `"/home/amanda/Documents/hw6/names.txt"`
   - `"data/numbers.txt"` or `"/home/amanda/Documents/hw6/data/numbers.txt"`
   - There is only one legal way to refer to this file: by its absolute path, `"/home/amanda/download/saved.html"`

7.
```
input: 6.7      This file has
input:  several input lines.
input:
input:   10 20  30   40
input:
input: test
6 total
```

8.
```
input: 6.7
input: This
input: file
input: has
input: several
input: input
input: lines.
input: 10
input: 20
input: 30
input: 40
input: test
12 total
```

9. using hasNextInt and nextInt:

```
0 total
```

using hasNextDouble and nextDouble:

```
input: 6.7
1 total
```

10.
```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class PrintMyself {
 5      public static void main(String[] args) throws FileNotFoundException {
 6          Scanner input = new Scanner(new File("PrintMyself.java"));
 7          while (input.hasNextLine()) {
 8              System.out.println(input.nextLine());
 9          }
10      }
11  }
```

11.
```
public static void printEntireFile() throws FileNotFoundException {
    Scanner console = new Scanner(System.in);
    System.out.print("Type a file name: ");
    String filename = console.nextLine();

    Scanner input = new Scanner(new File(filename));
    while (input.hasNextLine()) {
        System.out.println(input.nextLine());
    }
}
```

12. A PrintStream object is used to write to an external file. It has methods such as println and print.

13.
```
PrintStream out = new PrintStream(new File("message.txt"));
out.println("Testing,");
out.println("1, 2, 3.");
out.println();
out.println("This is my output file.");
```

14.
```
public static String getFileName() {
    Scanner console = new Scanner(System.in);
    String filename = null;
    do {
        System.out.print("Type a file name: ");
        filename = console.nextLine();
    } while (!(new File(filename).exists()));

    return filename;
}
```

15.
```
// re-prompts until file name is valid
public static void printEntireFile2() throws FileNotFoundException {
    String filename = getFileName();
    Scanner input = new Scanner(new File(filename));
    while (input.hasNextLine()) {
        System.out.println(input.nextLine());
    }
}
```

Chapter 7

1. first element: `numbers[0]`
   last element: `numbers[9]` or `numbers[numbers.length - 1]`

2.
```
int[] data = new int[5];
data[0] = 27;
data[1] = 51;
data[2] = 33;
data[3] = -1;
data[4] = 101;
```

3.
```
int[] odds = new int[22];
for (int i = 0; i < 22; i++) {
    odds[i] = i * 2 - 5;
}
```

4. {0, 4, 11, 0, 44, 0, 0, 2}

5.
```
public static int sumAll(int[] a) {
    int sum = 0;
    for (int i = 0; i < a.length; i++) {
        sum += a[i];
    }
    return sum;
}
```

6.
```
public static double average(int[] a) {
    double mean = 0.0;
    for (int i = 0; i < a.length; i++) {
        mean += a[i];
    }
    return mean / a.length;
}
```

   or, more concise version using solution from previous exercise:

```
public static double average2(int[] a) {
    return (double) sumAll(a) / a.length;
}
```

7. The code to print the arrays and to compare them doesn't work properly. Arrays can't be printed directly by println, nor can they be compared directly using the relational operators such as ==. These operations can be done correctly by looping over the elements of each array and printing/comparing them one at a time.

8.
```
public static boolean equal(int[] array1, int[] array2) {
    if (array1.length != array2.length) {
        return false;
    }

    for (int i = 0; i < array1.length; i++) {
        if (array1[i] != array2[i]) {
            return false;
        }
    }
    return true;
}
```

9.
```
int[] data = {7, -1, 13, 24, 6};
```

10. {20, 30, 40, 50, 60, 70, 80, 90, 100, 100}

11. {10, 10, 10, 10, 10, 10, 10, 10, 10, 10}

12.
```
for (int i = 0; i < data.length; i++) {
    System.out.println("Element [" + i + "] is " + data[i]);
}
```

13. a1: {26, 19, 14, 11, 10}
    a2: {1, 4, 9, 16, 25}

14. a1: {1, 3, -3, 13, -4, -24, -6, -14}
    a2: {1, 1, 2, 3, 5, 8, 13, 21}

15. {7, 3, 1, 0, 8, 18, 5, -1, 5}

16.
```
public static double averageLength(String[] strings) {
    int sum = 0;
    for (int i = 0; i < strings.length; i++) {
        sum += strings[i].length();
    }
    return (double) sum / strings.length;
}
```

17.
```
public static boolean isPalindrome(String[] array) {
    for (int i = 0; i < array.length / 2; i++) {
        if (!array[i].equals(array[array.length - 1 - i])) {
            return false;
        }
    }
    return true;
}
```

Chapter 8

1. Procedural programming treats a program as a sequence of actions or commands to perform. Object-oriented programming looks at a program as a group of interacting entities named objects that each keep track of related data and behavior.

2. An object is an entity that encapsulates data and behavior that operates on the data. A class is the blueprint for a type of objects, specifying what data and behavior each will have, and how to construct those objects.

3. Some objects we've used so far: String, Point Scanner, DrawingPanel, Graphics, Color, Font, Random, File

4. A data field is a variable that exists inside of an object. A parameter is a variable inside a method whose value is passed in from outside. Data fields have different syntax because they are usually declared with the private keyword and not in a method's header. A data field's scope is throughout the class, while a parameter's scope is just within that method.

5. An instance method doesn't have the static keyword. Also it may access or modify the data fields of that object using the this keyword. They're used differently because objects' methods are preceded by some variable's name and a dot.

6. An accessor provides the client access to some data in the object, while a mutator lets the client change the object's state in some way. Accessors' names often begin with "get" or "is", while mutators often begin with "set".

7. A constructor is a special method that creates an object and initializes its state. It's the method that is called when you use the new keyword. It is declared without a return type.

8. One problem is that the constructor shouldn't have the "void" keyword in its header, because constructors have no return type. The header should be:

   ```
   public Point(int x, int y) {
   ```

   Also, the data fields x and y shouldn't have their types redeclared in front of them. This is a bug and causes shadowing of the data fields. Here are the corrected lines:

   ```
   x = initialX;
   y = initialY;
   ```

9. Abstraction is the ability to focus on a problem at a high level without worrying about the minor details. Objects provide abstraction by giving us more powerful pieces of data that have sophisticated behavior without having to manage and manipulate the data directly.

10. Items declared public may be seen and used from any class. Items declared private may only be seen and used from within their own class. Objects' data fields should be declared private to provide encapsulation, so that external code can't make unwanted direct modifications to the fields' values.

11. To access private data fields, create accessor methods that return their values. For example, add a getName method to access the name field of an object.

12. To make the objects of your class printable, define a toString method in it.

13.
```
// Returns a String representation of this Point object.
public String toString() {
    return "java.awt.Point[x=" + this.x + ", y=" + this.y + "]";
}
```

14. The this keyword refers to the object on which a method has been called. It is used to access or set that object's data field values, or to call that object's other methods.

15.
```
// Constructs a Point object with the same x and y
// coordinates as the given Point.
public Point(Point p) {
    this.x = p.x;
    this.y = p.y;
}
```

    or,

```
// Constructs a Point object with the same x and y
// coordinates as the given Point.
public Point(Point p) {
    this(p.x, p.y);
}
```

16. The == operator doesn't work with objects because it just compares references to see whether they point to exactly the same object. It doesn't compare two objects to see whether they have the same state, which is usually what you want. If you want your objects to be compared correctly, define an equals method in your class and use equals instead of == in the client code.

17.
```
// Returns this Stock's number of shares purchased.
public int getShares() {
    return this.numShares;
}

// Returns this Stock's symbol value.
public String getSymbol() {
    return this.symbol;
}

// Returns this Stock's total cost.
public double getTotalCost() {
    return this.totalCost;
}
```

Chapter 9

1. Code reuse is the ability to write a single piece of code that can be used many times in different programs and contexts. Inheritance is useful for code reuse because it allows you to write a class that captures common useful code and then extend that class to add more features and behavior to it.

2. Overloading a method is creating two methods in the same class, that have the same name but different parameters. Overriding a method is creating a new version of an inherited method in a subclass, with identical parameters but new behavior to replace the old.

3. The following statements are legal:
    - `Vehicle v = new Car();`
    - `Vehicle v = new SUV();`
    - `Car c = new SUV();`
    - `SUV s = nev SUV();`

4.
```
B 2
A
A 1

D 2
C
C 1

A 2
A
A 1

A 2
C
C 1
```

5.
```
flute
shoe 1
flute 2

flute
blue 1
flute 2

moo
moo 1
moo 2

moo
blue 1
moo 2
```

6.
```
moo 2
blue 1
moo

moo 2
moo 1
moo

flute 2
shoe 1
flute

flute 2
blue 1
flute
```

7.
```
squid
creature 1
tentacles

BIG!
spout
creature 2

ocean-dwelling
creature 1
creature 2

ocean-dwelling
warm-blooded
creature 2
```

8.
```
creature 2
ocean-dwelling
creature 1

tentacles
squid
creature 1

creature 2
ocean-dwelling
warm-blooded

creature 2
BIG!
spout
```

9. The this keyword refers to the current object, while the super keyword refers to the current class's superclass. Use the super keyword when calling a method or constructor from the superclass that you've overridden, and use the this keyword when accessing your object's other data fields, constructors, and methods.

10. Freshman can call the setAge method but cannot directly access the name or age data fields from Student.

11. Two solutions are shown.

```
// Constructs a 3D Point at the origin of (0, 0, 0).
public Point3D() {
    super(0, 0);
    this.z = 0;
}

// Constructs a 3D Point at the origin of (0, 0, 0).
public Point3D() {
    this(0, 0, 0);
}
```

12. Extending a class causes your class to inherit all methods and data from that class. Implementing an interface forces you to write your own code to implement all the methods in that interface.

13. The code for class C must contain implementations of the methods m1 and m2 to compile correctly, because C claims to implement the I interface.

14. The interface is incorrect because interfaces can't declare data fields or write bodies for methods. The following is a correct interface:

```
1  import java.awt.*;
2
3  // Represents items that have a color that can be retrieved.
4  public interface Colored {
5      public Color getColor();
6  }
```

15.
```
1  // Represents a point with a color.
2
3  import java.awt.*;
4
5  public class ColoredPoint extends Point implements Colored {
6      private Color color;
7
8      // Constructs a new colored point with the given
9      // coordinates and color.
10     public ColoredPoint(int x, int y, Color color) {
11         super(x, y);
12         this.color = color;
13     }
14
15     // Returns this point's color.
16     public Color getColor() {
17         return color;
18     }
19 }
```

16. An abstract class is a class intended to be used only as a superclass for inheritance. It's like a normal class in that it can have data fields, methods, constructors and so on. It's different from a normal class in that it can have abstract methods, which are like methods of an interface because only their headers are given but not their bodies. It's also different from a normal class because it can't be instantiated (used to create objects).

17. One good design would be to have an abstract superclass named Movie with data such as name, director, and date. There would be subclasses of Movie to represent the particular movie types such as Drama, Comedy, Documentary. Each subclass would store its specific data and behavior.

Chapter 10

1. An ArrayList is a structure that stores a collection of objects inside itself as elements. Each element is associated with an integer index starting from 0. You should use an ArrayList instead of an array if you don't know how many elements you'll need in advance, or if you plan to add and remove items from the middle of your dataset.

2. ```
   ArrayList<String> list = new ArrayList<String>();
   list.add("It");
   list.add("was");
   list.add("a");
   list.add("stormy");
   list.add("night");
   ```

   The list's type is `ArrayList<String>` and its size is 5.

3. ```
   list.add(3, "dark");
   list.add(4, "and");
   ```

4. ```
   list.set(1, "IS");
   ```

5. ```
   for (int i = 0; i < list.size(); i++) {
       if (list.get(i).indexOf("a") >= 0) {
           list.remove(i);
           i--;    // so the new element i will be checked
       }
   }
   ```

6. ```
   ArrayList<Integer> numbers = new ArrayList<Integer>();
   for (int i = 0; i < 10; i++) {
       numbers.add(2 * i);
   }
   ```

7. ```
   public static int maxLength(ArrayList<String> list) {
       int max = 0;
       for (int i = 0; i < list.size(); i++) {
           String s = list.get(i);
           if (s.length() > max) {
               max = s.length();
           }
       }
       return max;
   }
   ```

8. ```
   System.out.println(list.contains("IS"));
   ```

9. ```
   System.out.println(list.indexOf("stormy"));
   System.out.println(list.indexOf("dark"));
   ```

10. ```
    for (String s : list) {
        System.out.println(s.toUpperCase());
    }
    ```

11. The code throws a ConcurrentModificationException because it is illegal to modify the elements of an ArrayList while for-eaching over it.

12. The code doesn't compile because primitives cannot be specified as type parameters for generic types. The solution is to use the "wrapper" type Integer instead of int. Change the line declaring the ArrayList to the following:

    ```
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    ```
    ```

13. A wrapper class is one whose main purpose is to act as a bridge between primitive values and objects. An Integer is an object that holds an int value. Wrappers are useful to allow primitive values to be stored into collections.

14. To arrange an ArrayList into sorted order, call the `Collections.sort` method on it. For example, if your ArrayList is stored in a variable named list:

    ```
    Collections.sort(list);
    ```

    For this to work, the type of objects stored in the list must be Comparable.

15. A natural ordering is an order for objects of a class where "lesser" objects come before "greater" ones, as determined by a procedure called the class's comparison function. To give your own class a natural ordering, declare it to implement the Comparable interface and define a comparison function for it by writing an appropriate compareTo method.

16. ```
    n1.compareTo(n2) > 0
    n3.compareTo(n1) == 0
    n2.compareTo(n1) < 0
    s1.compareTo(s2) < 0
    s3.compareTo(s1) > 0
    s2.compareTo(s2) == 0
    ```

17. ```
    Scanner console = new Scanner(System.in);
    System.out.print("Type a name: ");
    String name1 = console.nextLine();
    System.out.print("Type a name: ");
    String name2 = console.nextLine();

    if (name1.compareTo(name2) < 0) {
        System.out.println(name1 + " goes before " + name2);
    } else if (name1.compareTo(name2) > 0) {
        System.out.println(name1 + " goes after " + name2);
    } else {  // equal
        System.out.println(name1 + " is the same as " + name2);
    }
    ```

18. ```
    Scanner console = new Scanner(System.in);
    System.out.print("Type a message to sort: ");
    String message = console.nextLine();

    ArrayList<String> words = new ArrayList<String>();
    Scanner lineScan = new Scanner(message);
    while (lineScan.hasNext()) {
        words.add(lineScan.next());
    }

    System.out.print("Your message sorted: ");
    Collections.sort(words);
    for (String word : words) {
        System.out.print(word + " ");
    }
    System.out.println();  // to end the line of output
    ```
    ```

Chapter 11

1. You should use a LinkedList when you plan to do many adds and removes from the front or back of the list, or when you plan to make many filtering passes over the list in which you remove certain elements.

2. The code shown would perform better with an ArrayList, because it calls the get method many times using indexes in the middle of the list. This is a slow operation for a LinkedList.

3. An iterator is an object that represents a position within a list and the ability to view or make changes to the elements at that position. Iterators are often used with linked lists because they retain the position in the list so that we don't have to call expensive list methods like get, add, or remove many times.

4.
```
public static int countDuplicates(LinkedList<Integer> list) {
    int count = 0;

    Iterator<Integer> i = list.iterator();
    int prev = i.next();

    while (i.hasNext()) {
        int next = i.next();
        if (prev == next) {
            count++;
        }
        prev = next;
    }

    return count;
}
```

5.
```
public static void insertInOrder(LinkedList<String> list, String value) {
    int index = 0;

    Iterator<String> i = list.iterator();
    String next = i.next();

    // advance until the proper index
    while (i.hasNext() && next.compareTo(value) < 0) {
        next = i.next();
        index++;
    }

    list.add(index, value);
}
```

6.
```
public static void removeAll(LinkedList<Integer> list, int value) {
    Iterator<Integer> i = list.iterator();
    while (i.hasNext()) {
        if (i.next() == value) {
            i.remove();
        }
    }
}
```

7.
```
public static void wrapHalf(LinkedList<Integer> list) {
    int halfSize = (list.size() + 1) / 2;
    for (int i = 0; i < halfSize; i++) {
        // wrap around one element
        int element = list.remove(list.size() - 1);
        list.add(0, element);
    }
}
```

8. An abstract data type defines the type of data a collection can hold and the operations it can perform on that data. Linked lists implement the "List" abstract data type.

9. The countDuplicates code is identical to the preceding answer, but the method's header should be changed to the following:

```
public static int countDuplicates(List<Integer> list) {
```

10. You would use a Set rather than a List if you wanted to avoid duplicates or to have fast searching ability.

11. A TreeSet should be used when you want to keep the data in sorted natural order. A HashSet should be used with non-Comparable types or when order doesn't matter, to get the fastest searching time.

12. Examine every element of a Set using an Iterator.

13. The set contains: [32, 90, 9, 182, 29, 12]

14. The set contains: [79, 8, 132, 50, 98, 86]

15. The set contains: [94, 4, 11, 84, 42, 12, 247]

16. To do a union, use the addAll method to add one set's contents to the other. To do an intersection, use the retainAll method to remove elements not in common between the two sets.

17.
```
Map<String, Integer> ageMap = new TreeMap<String, Integer>();
ageMap.put("Stuart", 85);
ageMap.put("Marty", 12);
ageMap.put("Amanda", 25);
```

18. Examine every key of a map by calling the keySet method and then iterating or for-eaching over that keySet. Examine every value of a map by calling the values method and then iterating or for-eaching over that collection of values, or by looking up each associated value using each key from the keySet.

19. The map contains: {17=Steve, 34=Louann, 15=Moshe, 2350=Orlando, 7=Ed, 5=Moshe, 27=Donald}

20. The map contains: {79=Seventy-nine, 8=Ocho, 132=OneThreeTwo, 50=Fifty, 98=Ninety-eight, 86=Eighty-six}

21. The following method implements the new behavior into the Count program:

```
public static void reverseMap(
        Map<String, Integer> wordCountMap) {

    Map<Integer, String> reverseMap =
            new TreeMap<Integer, String>();

    // reverse the original map
    for (String word : wordCountMap.keySet()) {
        int count = wordCountMap.get(word);
        if (count > OCCURRENCES) {
            reverseMap.put(count, word);
        }
    }

    // print the words sorted by count
    for (int count : reverseMap.keySet()) {
        String word = reverseMap.get(count);
    }
}
```

Chapter 12

1. Recursion is a technique where an algorithm is expressed in terms of itself. A recursive method differs from a regular method in that it contains a call to itself within its body.

2. A base case is a situation where a recursive method does not need to make a recursive call to solve the problem. A recursive case is where the recursive method does call itself. Recursive methods need both cases because the recursive case is called repeatedly until the base case is reached, stopping the chain of recursive calls.

3.
   - 1
   - 1, 2
   - 1, 3
   - 1, 2, 4
   - 1, 2, 4, 8, 16
   - 1, 3, 7, 15, 30
   - 1, 3, 6, 12, 25, 50, 100

4.
```
public static void writeChars(int n) {
    if (n == 1) {
        System.out.print("*");
    } else if (n == 2) {
        System.out.print("**");
    } else {
        System.out.print("<");
        writeChars(n - 2);
        System.out.print(">");
    }
}
```

5.
```
public static void stutterReverse(String s) {
    if (s.length() == 0) {
        return s;
    } else {
        char last = s.charAt(s.length() - 1);
        System.out.print(last);
        System.out.print(last);
        stutterReverse(s.substring(0, s.length() - 1));
    }
}
```

6. A call stack is the structure of information about all methods that have currently been called by your program. Recursion produces a tall call stack in which each recursive call is represented.

7. The new code would print the lines in their original order and not reversed.

8. The new code would cause infinite recursion, because each recursive call just makes another recursive call and doesn't progress toward the base case.

9. The second version of the pow method is more efficient than the first because it requires fewer recursive calls. Both versions are recursive.

10.
```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

11.
- 6
- 4
- 7
- 0
- 1

12. The base case if statement has a bug. It should test for numbers less than 10, not greater. The following is the correct line:

```
if (n > 10) {
```

13. A fractal is an image that is recursively constructed to contain smaller versions of itself. Recursive methods are useful when drawing fractal images because they can elegantly express the recursive nature of the image.

14.
```
public static void drawHexegon(Graphics g, Point position, int size) {
    Polygon poly = new Polygon();
    poly.addPoint(position.x, position.y + size / 2);
    poly.addPoint(position.x + size / 3, position.y);
    poly.addPoint(twoThirdX, position.y);
    poly.addPoint(position.x + size, position.y + size / 2);
    poly.addPoint(twoThirdX, position.y + size);
    poly.addPoint(position.x + size / 3, bottomY);
    g.drawPolygon(poly);
}
```

Chapter 13

1. You can perform a sequential search over the array using a for loop, or you can sort the array using Arrays.sort and then perform a binary search over it using Arrays.binarySearch.

2. A sequential search should be used on an array of Point objects because they do not implement Comparable.

3. Arrays.binarySearch and Collections.binarySearch can be used successfully if the array or collection contains elements in a sorted order, either according to their natural ordering or the ordering of a Comparator.

• O(log n) • O(n)	• $O(n^2)$

5.
   - O(n)
   - $O(n^2)$
   - O(n)
   - O(n)
   - O(log B)
   - $O(n^3)$
   - O(n), where n is the number of lines or bytes in the file
   - O(1)

6. The runtime complexity of both sequential searches is O(n).

7. Binary search requires a sorted dataset because it uses the ordering to jump to the next index. If the elements are out of order, the search isn't guaranteed to find the target element.

8. A binary search of 60 elements examines at most 6 elements, because log 60 (when rounded up) equals 6.

9.
   - The algorithm will examine index 4 and return 4.
   - The algorithm will examine indexes 4 and 6. It will return 6.
   - The algorithm will examine indexes 4, 6, and 7. It will return 7.
   - The algorithm will examine indexes 4, 2, 1, and 0. It will return 0.

10. Because the input isn't sorted, the algorithm will examine indexes 4, 6, and 5. It will return -1.

11. The parameter array type should be changed to double. Also a new swap method will be needed that accepts a double[] as the first parameter. Here's the new code:

```
public static void selectionSort(double[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // find index of smallest element
        int smallest = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[smallest]) {
                smallest = j;
            }
        }

        swap(a, i, smallest);  // swap smallest to front
    }
}
```

12. A merge sort of 32 elements will generate 63 total calls to mergeSort and will perform the merge operation 31 times.

13. Java's sorting methods use quicksort for primitive data because it is very fast. They use merge sort for objects because it is a stable sort with well-bounded worst case performance.

14.
- After 5 passes of selection sort: {1, 2, 3, 4, 5, 11, 9, 7, 8, 10}
- The levels of merge sort:

```
{7, 2, 8, 4, 1, 11, 9, 5, 3, 10}
{7, 2, 8, 4, 1}, {11, 9, 5, 3, 10}
{7, 2}, {8, 4, 1}, {11, 9}, {5, 3, 10}
{7, 2}, {8, 4, 1}, {11, 9}, {5, 3, 10}
{7}, {2}, {8}, {4, 1}, {11}, {9}, {5}, {3, 10}
               {4}, {1},               {3}, {10}

{2, 7},   {8}, {1, 4}, {9, 11}, {5}, {3, 10}
{2, 7},   {1, 4, 8},   {9, 11}, {3, 5, 10}
{1, 2, 4, 7, 8},       {3, 5, 9, 10, 11}
{1, 2, 3, 4, 5, 7, 8, 9, 10, 11}
```

Chapter 14

1. Package `javax.swing` contains Java's Swing GUI classes. The statement to import this package is:

```
import javax.swing.*;
```

2.
```
String response = JOptionPane.showInputDialog(null,
        "What's your age, cowboy?");
int age = Integer.parseInt(response);
if (age < 40) {
    JOptionPane.showMessageDialog(null, "Only " + age +
            " ... still a young'n.");
} else {
    JOptionPane.showMessageDialog(null, "You're " +
            age + " ... howdy, old timer!");
}
```

3. A component is an onscreen item such as a button or text field. A frame is different from other components because it is a first-class citizen of the user's operating system. Frames hold other components inside them.

4. Some properties of frames are size, location, and title. The following code creates a frame and sets these properties:

```
JFrame frame = new JFrame();
frame.setSize(new Dimension(400, 300));
frame.setLocation(new Point(20, 10));
frame.setTitle("My favorite frame");
frame.setVisible(true);
```

5.
   - JColorChooser
   - JOptionPane
   - JComboBox
   - JButton
   - JRadioButton

6.
```
JButton b1 = new JButton("Click me");
b1.setBackground(Color.GREEN);

JButton b2 = new JButton("Do not touch!");
b2.setBackground(Color.YELLOW);
```

7. A GridLayout is used. The following code performs the layout:

```
frame.setLayout(new GridLayout(3, 2));
frame.add(b1);
frame.add(b2);
frame.add(b3);
frame.add(b4);
frame.add(b5);
```

8. A FlowLayout is used. The following code performs the layout:

```
frame.setLayout(new FlowLayout());
frame.add(b1);
frame.add(b2);
frame.add(b3);
frame.add(b4);
frame.add(b5);
```

9. A BorderLayout is used. The following code performs the layout:

```
frame.setLayout(new BorderLayout());
frame.add(b1, BorderLayout.NORTH);
frame.add(b2, BorderLayout.WEST);
frame.add(b3, BorderLayout.CENTER);
frame.add(b4, BorderLayout.EAST);
frame.add(b5, BorderLayout.SOUTH);
```

10. An event is an object representing a user's interaction with a graphical component. The ActionListener interface is used to handle events.

11. To handle an event, you must write a class that implements the ActionListener interface. It must contain a method named actionPerformed. You must call the addActionListener on the component in question to attach your new listener to it.

12.
```
1  import java.awt.event.*;
2  import javax.swing.*;
3
4  public class GreetingListener implements ActionListener {
5      public void actionPerformed(ActionEvent event) {
6          JOptionPane.showMessageDialog(null,
7                  "Greetings, Earthling!");
8      }
9  }
```

13. When doing 2D graphics, extend the JPanel class and write a paintComponent method.

14.
```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class RedCirclePanel extends JPanel {
5      public void paintComponent(Graphics g) {
6          super.paintComponent(g);
7          g.drawOval(10, 30, 50, 50);
8      }
9  }
```

15. A Timer is an object that can fire an ActionListener event at regular intervals. Timers can be used to animate panels by having their ActionListener change the state of the panel in some way and then repaint it.

16.
```
1  // An animated panel that draws a red and blue circle.
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class RedCirclePanel2 extends JPanel {
8      private boolean red;
9
10     public RedCirclePanel2() {
11         red = true;
12         Timer timer = new Timer(1000, new ColorListener());
13         timer.start();
14     }
15
16     public void paintComponent(Graphics g) {
17         super.paintComponent(g);
18         if (red) {
19             g.setColor(Color.RED);
20         } else {
21             g.setColor(Color.BLUE);
22         }
23         g.drawOval(10, 30, 50, 50);
24     }
25
26     private class ColorListener implements ActionListener {
27         public void actionPerformed(ActionEvent e) {
28             red = !red;
29             repaint();
30         }
31     }
32 }
```

Stuart Reges
Marty Stepp