Java™ Programming

Lyle A. McGeoch
Amherst College

2012 Edition, updated

Contents

Preface to the updated 2012 Edition

This is an updated version of 2012 edition, incorporating a number of updates to the chapter on graphics and some minor fixes. I expect to add more material on graphics and data structures in the coming months.

This book remains a work-in-progress. It is not ready for publication, but should serve as a good supplement to our class lectures. Comments are always welcome. Send any corrections or suggestions to me at `lam@cs.amherst.edu`.

Students should use Oracle's Java API documentation, available at

`http://download.oracle.com/javase/6/docs/api/`

for definitive documentation of the classes introduced in this book. I have tried to describe the most important features of the most important classes, but I've covered only a tiny fraction of what it available to programmers who are willing to explore.

Errata to this edition will be listed on book's website:

`https://www.amherst.edu/people/facstaff/lamcgeoch/javabook`

Enjoy!

Lyle A. McGeoch
Amherst College
August 2012

Chapter 1

Introduction

Computers are everywhere in today's world. Business, science, and education have long depended on high-speed computation. The PC revolution has brought computers into our homes, and the Internet has given us easy access to resources around the world.

As everyone knows, computers can't think. They excel in mindlessly following detailed instructions at very great speed. Over the last few decades, computers have become smaller, cheaper, and vastly more powerful. These developments, together with insights about how to make it easier for people to interact with machines, have brought profound changes in the ways we use computers. What hasn't changed is the need for humans to write the programs, the instructions that the computers must follow.

This book is a first introduction to programming, based on the computer language Java. This language was developed in the early 1990s by Sun Microsystems, and it has received much attention because it facilitates development of graphical, interactive web pages in a way that is largely independent of particular hardware. It turns out that Java is also an ideal first programming language, with features that are both simple and powerful.

Programming is essentially the task of *problem solving*. A problem is a task such as "Take a list of numbers and determine the average value." The solution to a problem isn't a particular answer, such as 5, but rather a strategy for accomplishing the task. When we express the solution as detailed instructions in a particular computer language, we have a program.

Programming can be a lot of fun. It's very satisfying when you succeed in getting a computer to perform a complex task. Once you know how to program, you'll probably find many occasions to write short programs for tasks you'd rather not do by hand.

Programming is also a good way to begin learning about the field of *computer science*, which examines the ways data can be organized and manipulated. Computer scientists study questions such as: What problems have efficient solutions? What problems will never be solved by computers? How can hardware and software work together in a multiprocessor system? How can secure commerce be handled on a network? Can we write programs that are provably correct? What computer language features will allow programmers to think in useful new ways? How can we quickly search vast sets of possibilities (such as in chess) without making too many mistakes? How can we write programs that essentially allow machines to learn on their own?

1.1 Applets and Applications

There are two kinds of Java programs, *applets* and *applications*.

Applets are graphical programs that interact with users via menus, buttons, scrollbars, text fields and similar features. Sometimes they use animation. Figure 1.1 shows an applet version of a calendar program. The user simply enters the

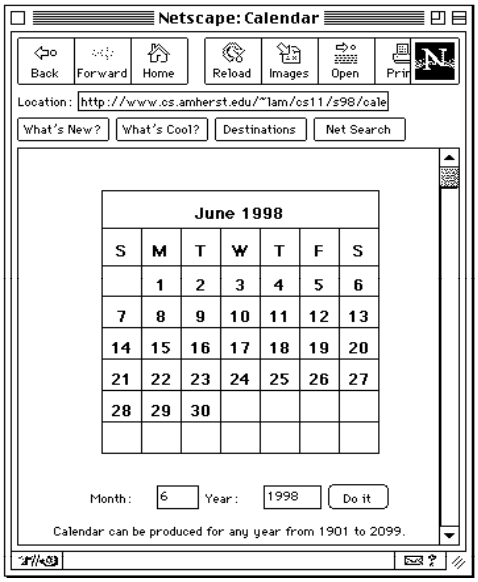

Figure 1.1: An applet for calendars

desired month and year in the text fields and clicks the "Do it" button.

Applets are similar to the standard kind of program that runs on a personal computer. One significant feature sets them apart: applets can be incorporated in

web pages. When a user (with an appropriate browser) visits a web page with an applet, the applet is loaded onto the user's machine and runs. Security mechanisms prevent the program from causing mischief on the the user's machine. Applets offer a powerful way for programmers to make programs available to users around the world.

Applications are programs that are designed to be run locally on a computer, without the use of a browser. Some applications use graphics, while others use a simpler approach in which the user and computer interact by typing in a window. For example, you might have the following "conversation" with a computer.

```
> java Calendar
Enter the year [1901-2099]: 2004
Enter the month [1-12]: 9

      September 2004
 S   M   T   W   T   F   S
             1   2   3   4
 5   6   7   8   9  10  11
12  13  14  15  16  17  18
19  20  21  22  23  24  25
26  27  28  29  30
```

I've used a typewriter font for things the computer types and italics for things the user types. The marker `>` is a *prompt* typed by the computer, indicating that the computer is waiting for a command. By typing *java Calendar*, you're asking for the computer to run a program called *Calendar*. When the program runs, it asks for a month and a year, and then it produces the calendar for the given month.

This is the traditional kind of program, based on a *command line* interface. Even though the interaction with the user is a simple conversation, applications can perform enormously complex tasks, perhaps using files stored on disk or perhaps using resources from across the Internet. Until about 1980, almost all programs used a command line interface.

For most programming tasks, you'll probably want to begin by writing an application that interacts with the user in a simple way. This allows you to concentrate on the algorithmic aspects of your program, deciding how to actually solve the task, while leaving the interface for later. Once you have a working application, you can add graphics or even turn your program into an applet. Most of this book will concentrate on applications.

1.2 Computers and Machine Code

There are three essential elements to a computer: the *memory*, *devices* and the *processor*.

Let's start with *memory*. Computers depend on their ability to manipulate large amounts of information. When you run a program, part of the computer's memory is used to hold *data*, the information being manipulated, and part of the memory is used to hold *instructions*, which tell how that data should be used. For example, if you are using a word processor, the computer's memory probably holds both your document and the word processing program itself. Today's computers have huge memories, at least compared to machines from a few years ago. New PCs being sold today typically have at least 256 megabytes of memory, meaning that they can hold over 256 million characters of information.

Devices allow the computer to interact with the outside world. Typical devices include the keyboard, mouse, and screen. Without such devices, the computer would be useless. Many modern computers use other audio and visual devices, and almost all use network devices such as Ethernet cards. Hard disks and removable disks are also devices. Disks are usually much larger and slower than the ordinary memory and are used for longer-term storage of information. New PCs typically have disks that are large enough to hold billions of characters of information!

The *processor* is the heart of the computer. It is a unit that can loads instructions and data from memory and that can *execute* the instructions. There are many kinds of instructions; among the most important are ones that specify that an operation such as addition should be performed, ones that tell how information should be moved to or from memory, and ones that make decisions about what the processor should do next. Together, these instructions allow the computer to do long calculations without human intervention.

Writing programs in *machine code*, the real language used by computers, is painful. Consider the example in Figure 1.2, which is shown in assembly language, a human-readable form of machine code. In the early days of computers, all programs were written using this kind of language, easily understood by computers but difficult for people. Eventually *high-level languages* were developed to permit people to express their ideas more easily. Fortran, COBOL, Pascal, Ada, C, C++, Lisp, and Java are all examples of high-level languages. Special programs called *compilers* are used to convert programs in high-level languages to the low-level machine code required by computers.

```
.LL2:
        ld [%fp-20],%o0
        cmp %o0,99
        ble .LL5
        nop
        b .LL3
        nop
.LL5:
        ld [%fp-24],%o0
        ld [%fp-20],%o1
        add %o0,%o1,%o0
        st %o0,[%fp-24]
.LL4:
        ld [%fp-20],%o0
        add %o0,1,%o1
        st %o1,[%fp-20]
        b .LL2
        nop
.LL3:
        sethi %hi(.LLC0),%o1
        or %o1,%lo(.LLC0),%o0
        ld [%fp-24],%o1
        call printf,0
        nop
```

Figure 1.2: A program in assembly language

1.3 Creating a Java Program

Figure 1.3 shows a small Java program. When you run this program, it simply writes the words "Hello world!" on the computer's screen. Notice how simple this program is compared to the one in assembly language.

There are three main steps in creating a Java program: *editing*, *compiling*, and *running*. Figure 1.4 illustrates the relationship of these steps. You begin by using an editor, a program similar to a word processor, to enter the program and to save it as a file on your computer's disk. The program shown in Figure 1.3 is called `Hello`; it would be saved in a file called "`Hello.java`". The `.java` suffix indicates that the file contains a Java program. After you've saved your Java program, you compile it to produce a `.class` file. The `.class` file contains *Java virtual machine code*, a form of machine code. Once you've created the `.class` file, your program is ready to run. You run your program by using the *Java virtual machine*. The virtual machine, which is itself a program, carries out the instructions in your program.

```
1   public class Hello {
2
3     // Here's our first program
4     //                          L. McGeoch, 6/2004
5
6     public static void main (String[] args) {
7
8       System.out.println ("Hello world!");
9     }
10  }
```

Figure 1.3: A little program

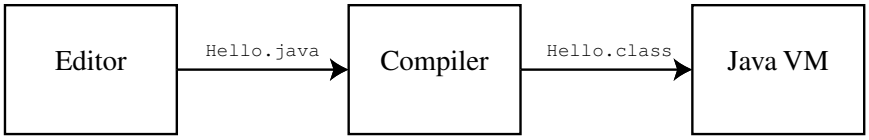

Figure 1.4: Creating a Java program

Beyond the basic steps of editing, compiling and running, there is also *debugging*. If there are errors in your program, you will need to go back and fix the `.java` file, and then recompile and rerun. We'll talk more about debugging and errors in section 1.6.

Many students learn programming on Unix computer systems. On a Unix system, a programmer usually uses separate programs for editing, compiling, and running. Other students use an IDE, an integrated development environment, such as Eclipse. An IDE combines an editor, compiler, runner, and debugging tools, and offers a powerful tool for program development. Regardless of the software you use to develop programs, the essential steps remain the same.

1.4 Algorithms

Programs are based on *algorithms*, methodical strategies for solving problems. Algorithms need not be expressed in a particular computer language, but they must be clear and unambiguous. Let's look at an example.

Suppose someone tells you that they've chosen a number between 1 and 100 and asks you to guess it. When you make a guess, they'll tell you if you've found the number, if you're high, or if you're low.

Here's one strategy you might use:

1. Choose 1 as the number to try.
2. Ask if that's the right number.
3. If the guess is right, celebrate and quit playing.
4. Otherwise, add one to your number and go back to step 2.

Probably you think that this is a pretty lousy way to play the guessing game. That's true, because it doesn't use the high/low information in a very effective way. Nevertheless, it's a methodical strategy, an algorithm. You could write a Java program based on this algorithm.

Here's a better approach:

1. Remember 1 as the "lowest possible number" and 100 as the "highest possible number."
2. Guess a number midway between the lowest and highest possible numbers. Call this number m.
3. If the guess is right, celebrate and quit playing.
4. If it's high, the number must be less than m. Change your record of the "highest possible number," so that it equals $m - 1$. Now go back to step 2.
5. Otherwise the guess is low. The number must be more than m. Change your record of the "lowest possible number," so that it equals $m + 1$. Go back to step 2.

This is probably close to the way most people play this game. This too is an algorithm that could be the basis of an actual program. Intuitively it's more efficient that the first algorithm, and there is a mathematical sense in which this is provably true. Much of the fun and challenge of programming is thinking about strategies in this way.

1.5 Top-Down Design

The calendar program we used earlier was based on the following steps:

- Type a message asking the user to enter a year. Read a number from the keyboard. If it's between 1901 and 2099, keep going, otherwise quit with an error message.
- Type a message asking the user to enter a month. Read a number from the keyboard. If it's between 1 and 12, keep going, otherwise quit with an error message.
- Type the banner line, giving the name of the month and year, followed by a line labelling the different days of the week.
- Determine how many days are in the given month.
- Determine the day of the week on which the given month begins.
- For each week w, where w is a number from 1 to 6, write the w^{th} line of the calendar.

While it is probably clear that this is a correct approach to the overall task of producing a calendar, it leaves some questions unresolved. How do we determine the number of days in a given month? How do we know the day on which a given month starts? How exactly do we print the w-th line of the calendar?

Let's focus on one of the subtasks, determining the number of days in the month. Here's a strategy:

- If the month is 4, 6, 9, or 11, the answer is 30. ("Thirty days hath September...").
- Otherwise, if the month is 1, 3, 5, 7, 8, 10, or 12, the answer is 31.
- Otherwise, if it's a leap year, the answer is 29.
- Otherwise, the answer is 28.

This looks right, but it leaves open yet another question: determining if a year is a leap year. This is another subtask to be solved. By systematically dividing the problem into subtasks and then solving the subtasks, it is possible to solve the whole problem of producing a calendar. This illustrates the idea of *top-down* design, where a top-level problem is reduced to smaller and smaller problems.

In Java, a *method* is used to describe the solution to a subtask. For example, Figure 1.5 shows a method called `getDaysInMonth` that determines how many days are

```
public static int getDaysInMonth (int month, int year) {

   if (month == 4 || month == 6 || month == 9 || month == 11)
     return 30;
   else if (month == 1 || month == 3 || month == 5 || month == 7 ||
            month == 8 || month == 10 || month == 12)
     return 31;
   else if (isLeapYear(year))
     return 29;
   else
     return 28;
 }
```

Figure 1.5: A method to calculate the number of days in a month.

in a given month. This method depends on another method, `isLeapYear`, in order to do its work. We'll talk much more about the details of this code later. Methods are sometimes known as *functions*, *procedures* or *subroutines* in other programming languages.

One key to good algorithm design, and good programs, is an appropriate decomposition of the overall problem into subtasks. If good choices are made, the methods that solve the subtasks can be fairly simple, leading to a clean, understandable program. While it is possible to write programs without decomposition, they usually become confusing and unwieldy very quickly.

Java also supports the use of *classes* in developing well organized programs. Classes, together with *inheritance*, are the basis of *object-oriented programming*, an approach to programming that is useful in producing clean, reusable code. We'll talk much more about classes later in the book.

1.6 Errors

Errors happen. There are three main kinds of errors that you'll encounter when you program:

- *Compilation errors.* These are errors in following the precise rules of the Java language. For example, semicolons are usually used to separate distinct steps of a program. If you leave one out, the program is meaningless, and it will fail when you try to compile it. You'll get error messages, which will be more or less helpful. When a compilation error occurs, you'll need to figure out what

the problem is, and then you'll need to go edit your program, fix the error, and recompile.

Compilation errors are the simplest kind of errors, because you can't avoid noticing them.

- *Run-time errors.* Sometimes your program will compile fine, but it will "crash" or fail in some way when you try to run it. For example, your program might try to divide a number by zero. This is an illegal operation, and it will cause an *exception*. Generally this means that your program will simply stop and issue an error message. You'll need to figure out what happened, fix the `.java` file, recompile, and rerun.

  Run-time errors are trickier than compilation errors, because an erroneous program might fail only part of the time.

- *Logical errors.* You might have a program that always compiles and never crashes. Is it right? Perhaps it is, but perhaps it doesn't do what you intended. For example, suppose you wanted a program to read a list of numbers and sort them. If the program prints them out in the wrong order, or (worse yet) if it prints out different numbers, you have a logical error.

  Logical errors are the most difficult kind to handle, because the program's results might look right.

Errors can often be detected and eliminated with careful testing. Methods can often be tested individually. Confidence in the correctness of individual components of a program can then lead to confidence in the overall correctness. We'll talk about how to program defensively, so that you can catch errors early.

1.7 Why Java?

There are many reasons why Java is a good first programming language:

1. Java is a safe language, in the sense that many errors are detected that might be ignored in another language. This greatly simplifies the task of debugging.

2. Java is object-oriented. As mentioned earlier, classes and inheritance provide a powerful organizational tool.

3. Java has an extensive library of built-in methods and classes, which programmers can use in their own work. Java's facility for handling strings is much better than that in C or Pascal.

4. Java is portable. If a Java program works on one computer, it should be able to work on any computer that handles Java.

5. Java uses garbage collection, a mechanism for recovering memory that is no longer needed. This simplifies programming.

6. Java supports graphics and can be used in web pages.

7. Java is fairly simple.

Java is not perfect. In order to achieve safety and portability, Java is *interpreted*, which essentially means that every step is managed by a master program, the Java Virtual Machine (VM). This means that Java is usually slower than languages such as C, Pascal, or C++. Garbage collection also affects speed. Recent versions of the Java VM have incorporated changes that improve the speed of both interpretation and garbage collection, and the speed gap between Java and other languages has been greatly reduced. Java is also still evolving, meaning that different computers might have different versions, reducing portability.

Despite these concerns, Java is a fine first language. The safety features and library make it easy for programmers to get started, focusing on algorithms and program organization. Many students will want to go on to learn C++, which has many similarities to Java. C++ is faster than Java and offers different opportunities for creating reusable code, but lacks some of Java's safety and simplicity.

This book is based on Java 6.

1.8 Conclusion

This book is intended to help you learn how to write programs that are clear and correct. We will especially focus on the following issues:

- Top-down design. Decomposition of a task into subproblems is a key first step in solving the overall problem. Proper use of methods and classes can greatly simplify programming.

- Algorithms. It's important to find the right general approach to solving a problem. Recall the two algorithms given earlier for trying to guess another person's secret number. While it's easy to simply try every number, it's far better to use a more sophisticated approach.

- Error prevention and detection. Programs with errors are useless. Actually they're worse than useless if you depend on them in some critical way. Careful testing at each step of program development can help you produce correct programs.

- Style and documentation. Much of the "style" of a program relates to the way methods and classes are used to lead to an overall solution. There are aspects of style that go beyond this. As we'll see in the chapters ahead, the choice of names, the use of spacing, and the use of comments (intended for humans, not the computer), can make the difference between a confusing program and one that is clear and understandable. If you pay attention to these ideas, your programs will be much easier for others to read, understand, evaluate, and maintain. Obviously this is critical when working on a group project. Good style and documentation will also make it easier for you to understand your own programs, even while working on them.

Chapter 2

Getting Started

Let's begin by looking at an expanded version of the "Hello world" program from Chapter 1. To simplify discussion, each line of the program is numbered. These numbers are not part of the actual program.

```
1   public class HelloAgain {
2
3       // Here's another program, with an easy calculation.
4       //                          L. McGeoch, 9/2004
5
6       public static void main (String[] args) {
7
8           int i = 3;
9           i = i * 2;
10
11          System.out.println ("Hello again!");
12          System.out.println ("Three times two is " + i);
13      }
14  }
```

Figure 2.1: Another short program

If you run this program, the following output will appear on the screen:

```
Hello again!
Three times two is 6
```

Recall that programs are based on *methods*. There might be many methods, each handling part of the overall task. Every program (unless it's an applet) has

a method called `main`. When you run a program, execution begins with the main method. The first line of a main method, its *method header*, should always be

```
public static void main (String[] args) {
```

Don't worry about understanding the precise meaning of this line yet. The main method's header is on line 6. The method's *body* begins with the left brace, `{`, on line 6 and continues through the right brace, `}`, on line 13. Within the body are the *statements*, the individual steps of the method. When the program runs, the statements are executed in order, beginning at the top.

Line 8 is a *declaration* that says a *variable* called `i` should be created. A variable is a piece of computer memory capable of holding some kind of information. In this case, `i` is declared to be an *int*, meaning that it can hold any integer (at least up to a very large limit that we'll discuss later). Variable `i` initially contains the integer 3.

Line 9 is an *assignment statement* It computes a value, `i * 2`, and puts the result into variable `i`. The asterisk, `*`, is the multiplication symbol in Java. This statement occurs after the declaration and initialization, so the effect is to change the value of variable `i` from 3 to 6.

Lines 11 and 12 are *method calls*. Each uses `System.out.println`, a method that is built into Java, to write to the computer's screen. One line of output simply contains the words "Hello again!", while the other contains the words "Three times two is " followed by the value contained in variable `i`.

Every method in Java is part of a *class*. Classes are a powerful mechanism for organizing programs, but for now you'll simply use them as containers for methods. The first line gives the name, `HelloAgain`, of the class. The actual class consists of everything between the left brace, `{`, at the end of the first line and the right brace, `}`, on line 14.

Lines 3 and 4 are *comments*, messages for human readers of a program. Comments are ignored by the computer when it processes the program, but it's important to use them to explain your work. Whenever you use a pair of consecutive slashes, `//`, the rest of the line is treated as a comment. Blank lines and spacing, except within quotation marks, don't affect the computer's handling of a program.

2.1 Editing, Compiling, and Running a Program

Section 1.3 described the basic steps involved in creating a program: editing, compiling, and running. Let's look in more detail at how these steps are done on a typical Unix system. Users of IDEs will need to consult their documentation or local experts.

The program we discussed above was based on a class called `HelloAgain`. Before you can use this program, it must be typed into a file called "`HelloAgain.java`". In general, the file must have precisely the same name as the class, with `.java` added as a suffix. Java usually distinguishes between upper and lower case, so a class `Fun` must be in `Fun.java`, not `fun.java` or `FUN.java`.

On many Unix systems, the text editor `emacs` is a good tool for entering programs. For example, you might type the command

```
emacs HelloAgain.java
```

to begin entering your program. Other systems will have other ways for you to enter programs. Your instructor, system manager, or computer center should be able to give you more information about how to do it on your system.

To compile your program, turning it into virtual machine code, you use the command

```
javac HelloAgain.java
```

If the compilation is successful, you'll get a file called "`HelloAgain.class`". If there's an error in your program, you'll get an *error message*, with a brief description of the problem. Perhaps you've misspelled something, left out punctuation, or put ideas together in an incorrect way. We'll discuss various messages and what they mean later. If you do get an error message, you need to figure out what the problem is, and then edit your program. This almost never means that you need to go retype the whole program; you just need to use the editor to fix whatever was wrong. You should never edit the `.class` file, because it contains material that looks like gibberish in an editor. This gibberish is simply the computer's version of your program.

Once you have successfully compiled your program, you can run it on the Java Virtual Machine. On a Unix system you use the command:

```
java HelloAgain
```

As noted above, our program will type:

```
Hello again!
Three times two is 6
```

Once you run your program, you may decide that it doesn't work correctly. You can then go change the program, and then compile and run again. *You never need to recompile unless you've changed your program.*

2.2 Identifiers

Class names and variable names are `identifiers`, names chosen by the programmer. An identifier can contain any sequence of letters or digits, beginning with a letter. The dollar sign, `$`, and the underscore, `_`, are considered letters. Upper-case letters and lower-case letters are distinct, so a variable `hi` is different from a variable `Hi`.

Here are some examples of legal identifiers:

```
hiThere  r2d2  mytemp  This_is  $$3
```

Here are some examples that are not legal:

```
2bOrNot2b  b+c
```

It is good style to choose class names that are meaningful and begin with an upper-case letter. Good examples are:

```
ChessMeister  TempCalc  Lab3  GameEditor
```

Variable names should be meaningful and should start with a lower-case letter. The upper-case versus lower-case rule helps readers distinguish different kinds of identifiers at a glance. Examples of good variable names are:

```
temp3  total  herAge  studentCount  prevMonth
```

Sometimes it is permissible to use very simple variable names, such as `i`, `j`, or `m3`. It is essential to use such names only in situations where the meaning is clear; otherwise a more descriptive name should be chosen.

Any word that is a *reserved word* can not be used as an identifier. (Most of these are *keywords*; others are names of *types*.) The Java reserved words are:

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

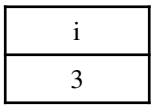

Figure 2.2: A graphic representation of a variable

Java has a large set of associated classes, known as the Java API. You can cause conflicts and create strange errors if you happen to give your class the same name as one of these special classes. Here are some of the names you should try to avoid:

```
String  System  Scanner  Object  Class
```

2.3 Variables

A program uses *variables* to store information that is used in a computation. Every variable has an identifier (its name), a type (the kind of information it can hold), and a value. Once created, a variable's name and type will never change, but its value may be modified.

For example, in the program in Figure 2.1, we used the statement

```
int i = 3;
```

This is a *declaration* that says that a variable `i` of type *int* should be created. Such a variable can hold any integer that is, ignoring sign, less than 2^{31}, about two billion. In this example, the declaration of `i` is combined with an *initialization* that gives the variable an initial value.

It's useful to think of a variable as a box, such as in Figure 2.2. The top part of the box gives its name, and the bottom part its value. In reality the box corresponds to a small piece of computer memory, which holds information that is stored and may be needed later. Here are several more examples:

```
int maxStudents = 100;        // initialization
int studentCount;             // no initialization
int myAge, hisAge, herAge;    // multiple variables
```

If no initial value is given for a variable, it is *uninitialized*. You can use comments to annotate declarations and explain the purpuse of the variables.

Another basic type is *double*. A double variable can contain a floating-point number, that is, a number that is not necessarily an integer. For example, you can make the following declarations:

```
double pi = 3.14159;
double gpa;                    // grade point average
double bankBalance = 0;
```

Variables of type double can be used to store huge numbers (up to about 10^{308}) or tiny numbers (as small as 10^{-324}), but they have limited precision. We'll talk more about this later. A double can be initialized to an integer value.

Technically, a variable of type double holds a *double-precision floating point number.* There's another type called *float* that is used for a *single-precision floating point number.* Variables of type float use less computer memory than doubles and can sometimes be manipulated faster. Unfortunately floats are less precise and are subject to more rounding errors. For this reason it's usually better to use doubles. We'll talk more about precision later.

2.4 Assignments and Expressions

A program can contain *assignment statements*, which compute a value and place it in a variable. For instance, you can write

```
myAge = hisAge + 1;
```

The effect of this statement is to take the value found in variable `hisAge`, add one to it, and store the result in variable `myAge`. Both `myAge` and `hisAge` must have already been declared as variables. In addition, `hisAge` must already contain a value, either because it was initialized when it was declared or because it received a value in a previous assignment.

In general, the form of an assignment statement is:

variable `=` *expression* `;`

This statement takes a value, determined by the expression, and saves it in the variable. The type of the expression must be compatible with the variable, in a way that we'll discuss shortly.

An *expression* is a value or a computation that yields a value. Here are some simple expressions:

```
i
3.4
i + (j*k)
(2+j) - (c/6)
```

Every expression has a type, for example int or double. Every variable can be used, by itself, as an expression. So you can write

```
i = j;
```

as long as `i` and `j` are both declared and have appropriate types. This copies the value in `j` into `i`. Numerical constants can also be used as expressions. For example, you can write

```
i = 4;              // 4 is an integer constant
j = -4;             // -4 is an integer constant
a = 3.0;            // 3.0 is a double constant
b = -2.1;           // -2.1 is a double constant
c = -3.2e3;         // an example of scientific notation
d = 6e-2;           // an example of scientific notation
f = 6.;             // a double constant equal to 6.0
g = -.4;            // a double constant equal to -0.4
```

As these examples show, negative numbers can be expressed by putting a negative sign immediately before the number. Positive numbers can be indicated with a plus sign, such as it `+34`, but the plus sign is usually omitted. A double constant can be given in scientific notation. In the examples above, `-3.2e3` means $-3.2 \cdot 10^3 = -3200.0$, while `6e-2` means $6 \cdot 10^{-2} = 0.06$. Either an `e` or `E` can be used to indicate scientific notation; there should be no space immediately before or after the letter.

Two expressions can be combined by using the arithmetic operators for addition (`+`), subtraction(`-`), multiplication(`*`), and division(`/`). You can also use parentheses to indicate an order of operations. For example, you can write

```
i = j + k;
b = 3 * 4.1;
a = i - (2.0 * j);
c = j / 3;
```

Now types becomes significant. If you do arithmetic on two ints, the result is an int. If you do arithmetic on two doubles, the result is a double. If you do arithmetic on an int and a double, the int is converted to a double, the operation is performed, and the result is a double.

For example, suppose you have `m = n - 2;`. If `n` is an int variable containing the value 4, then the expression yields the value 2, which is then assigned to variable `m`. If `n` is a double variable containing the value -3.2, the expression yields the value -5.2. If `n` is a double containing the value 3.0, the expression yields the value 1.0. The expression's type is double, even though its value happens to equal an integer.

These rules lead to some interesting consequences for the division operator, `/`. The expression `3 / 4.0` means that the integer 3 is converted to a double and then divided by 4.0, yielding the quotient 0.75. The expression `11 / 6` means that the integer 11 is divided by 6, yielding an integer. That integer is 1, the result of "whole number" division of 11 by 6, with rounding towards zero. This behavior sometimes causes unintended results. For example, you might try to write

```
celsius = (fahrenheit-32) * (5/9);          // error
```

in order to convert a temperature from Fahrenheit to Celsius. Unfortunately, the division will yield zero, and the result will always be zero. A correct way to write this might be

```
celsius = (fahrenheit-32) * (5/9.0);        // this works
```

Integer remainders can be computed with the `%` (modulus) operator. For example, `11 % 6` yields 5, the remainder that is obtained when the integer 11 is divided by the integer 6.

2.5 More on Assignments and Initializations

An int expression can be assigned to either an int or a double variable. A double expression can only be assigned to a double variable. This makes sense, because a double is "big enough" to hold either kind of number, while an int can't hold every double value. Suppose you have the following declarations:

```
int i, j, k;
double a, b, c;
```

The following sequence of statements is correct:

```
i = 3;
a = i + 5;              // a gets the value 8.0
b = 12 / a;             // b gets the value 1.5
c = i + b;              // c gets the value 4.5
c = c - b;              // c now gets the value 3.0
j = 12 / 3;             // j gets the value 4;
a = 12 / 3;             // a gets the value 4.0
b = 13 / 3;             // b gets the value 4.0!
```

The following statements are not legal:

```
i = 4.3;              // not legal
j = 4.0;              // not legal
k = a + 1.7;          // not legal
```

Remember, every variable that appears in an expression must have been declared and must have already been given a value, either when it was declared or by some assignment statement.

Any expression that can be used in an assignment can also be used in the initialization of a newly declared variable. For example, if `i` is a variable that has been declared and has a value, it is legal to write

```
double d = i*2;
```

All the usual rules about types apply, so this initialization will work if `i` is either an int or a double.

2.6 More on Arithmetic Expressions

As mentioned earlier, parentheses can be used to indicate the order in which operations can be applied. Furthermore, the minus sign can be used as a *unary operator* to obtain the negative of an expression. The expression

```
d = d * -(4.5/f);
```

means that 4.5 should be divided by `f`, the result should be negated and multiplied by `d`, and the result should be placed back in `d`.

Java has precedence rules that specify the order in which operators should be applied. Parentheses can be used to force a particular order. In the absence of parentheses, operations happen in the following order:

1. Negation: Unary minus signs are used to negate values.
2. `*`, `/`, and `%`: These are performed from left to right.
3. `+`, `-`: These are performed from left to right.

These rules correspond to the order usually used in ordinary arithmetic.

Consider the statement

```
d = e*f + g/h/i;
```

Multiplication and division take precedence over addition, and a sequence of these operators are handled left-to-right, so this is equivalent to:

```
d = (e*f) + ((g/h) / i);
```

Now consider this statement:

```
a = b * c+d * e;
```

The rules remain the same. Despite the misleading spacing, multiplication take precedence over addition, so it is equivalent to

```
a = (b*c) + (d*e);
```

Be sure that your spacing helps, rather than hinders, the understanding of your programs.

Here's a more complicated example:

```
d = e + g * h - (i * (j/k)) + m / -n / p;
```

Negation has highest precedence, so effectively this example is equivalent to:

```
d = e + g * h - (i * (j/k)) + m / (-n) / p;
```

Multiplication, division, and remainders come next, and a sequence of these operations is applied left-to-right, so the statement is equivalent to:

```
d = e + (g*h) - (i * (j/k)) + ((m/(-n)) / p);
```

The remaining additions and subtractions are handled from left to right, so `e` is added to `g*h`, then `i*(j/k)` is subtracted, and so on.

You should strive to write expressions (and programs!) that are as simple and clear as possible. Use parentheses as needed to make your meaning clear. And be sure that your spacing doesn't mislead the reader.

If you have a double and you want to turn it into an int value, you can do a *cast*. You do this by writing `(int)` before the expression to be converted. The fractional part of the double is effectively discarded. For example, you can write:

```
i = (int) 3.4;                  // expression equals 3
j = (int) (a / .61);
k = (int) -9.1;                 // expression equals -9
```

Casting has precedence equal to that of the unary minus sign, so you need to use parentheses if you want to convert an expression involving operators such as addition and multiplication.

2.7 Dangers with Numbers

If you have a numerical expression whose value is too large, *overflow* can result. For example, if you have:

```
int billion = 1000000000;
int value = billion * billion;
```

You'll actually end up with `value` equalling −1486618624. This happens because no int can hold a number larger than about 2 billion. In effect, the internal representation of a large number ends up being confused with that of some other number. The same thing can happen in the negative direction: numbers smaller than about −2 billion cannot be represented.

Floating-point values have different limitations. Overflow can occur, though at a much higher cutoff. The more significant problem is *precision*. If a person is asked to divide one by three, he or she can't write the answer down as a decimal number without losing part of the answer. The answer is close to 0.33333333, but that value is only approximate. Values of type double have about 15 decimal digits of precision. Interestingly, because computers are based on binary (base 2) arithmetic, numbers such as 1/5.0, which can be represented precisely in decimal, cannot be represented precisely in a computer. Because of imprecision, the value of `a / b * b` might not equal the value of `a`.

Every math student learns that it's impossible to divide a number by zero. If a Java program divides an int by zero, an `ArithmeticException` happens, causing the program to halt. (We'll talk about exceptions later.) If a double is divided by zero, no exception occurs, but the result is a special value, `NaN`, meaning *not a number*. You should always ensure that your programs don't try to divide numbers by zero!

2.8 A Second Program

Let's take a look at another program, shown in Figure 2.3. When you compile and run this program, the computer asks for a temperature to be entered in Fahrenheit degrees. It then prints the equivalent temperature in Celsius and Kelvin. For example:

```
Enter a temperature in Fahrenheit degrees: 56
That's 13.333333333333334 degrees Celsius,
or 286.4833333333333 degrees Kelvin.
```

Italics show what the the user types; the rest is typed by the computer.

```
1   import java.util.Scanner;
2
3   public class TempCalc {
4
5       // This program does temperature conversion.
6       //                          L. McGeoch, 9/2004
7
8       public static Scanner keyboard = new Scanner(System.in);
9
10      public static void main (String[] args) {
11
12          double fahrenheit;  // These variables contain
13          double celsius;     // temps in three different
14          double kelvin;      // systems.
15
16          System.out.print ("Enter a temperature in Fahrenheit degrees: ");
17          fahrenheit = keyboard.nextDouble();   // reads from the keyboard
18
19          celsius = (fahrenheit-32) * (5.0/9.0);
20          kelvin = celsius + 273.15;
21
22          System.out.println ("That's " + celsius + " degrees Celsius,");
23          System.out.println ("or " + kelvin + " degrees Kelvin.");
24      }
25  }
```

Figure 2.3: A program to do temperature conversion.

Let's look at the main method of this program. Lines 12 through 14 declare three variables, `fahrenheit`, `celsius`, and `kelvin`, without giving them initial values. Line 16 uses `System.out.print` to ask the user to enter a temperature. Line 17 reads a floating point number from the keyboard and places it in variable `fahrenheit`. Lines 19 and 20 apply the appropriate formulas to convert the temperature to the other temperature scales. Finally, lines 22 and 23 produce the output.

Most simple programs are structured just like this one. The program asks for information, it reads the information, it computes something, and it writes the answer.

2.9 Input and Output

Let's look more carefully at how a program can get information from a user. Two special statements must appear in any program that requires keyboard input. The statement

```
import java.util.Scanner;
```

must be at the very top of the program, even before the line giving the name of your class. The statement

```
public static Scanner keyboard = new Scanner(System.in);
```

must appear within your class but outside your main method. The program above includes these statements on lines 1 and 8; together the statements ensure that the program can read from the keyboard.

If a program has established access to the keyboard, it can read numbers with the method call `keyboard.nextDouble()`. The call waits for a number to be typed at the keyboard. That number could be a floating point number, such as `4.51` or `2.3e4`, or it could be an integer, such as `-23`. In any event, after the number is typed, `keyboard.nextDouble()` gives a value that can be used in some way. Typically, programmers will simply take the value and assign it to a variable. On the other hand, you can also write the following:

```
double twice;

System.out.print ("Enter a number: ");
twice = keyboard.nextDouble() * 2;
System.out.println ("Twice your number is " + twice);
```

You can use `keyboard.nextDouble()` anywhere where you might otherwise use a variable or expression of type double. It will always cause the computer to wait for a double to be entered at the keyboard.

If the computer is trying to read a number and the user types something else, such as the word "hi," an `InputMismatchException` occurs, which generally means that the program will stop. Later we'll see how you can write programs that are more robust, giving users the chance to correct errors.

Sometimes a program will need to read an integer instead of a double. In this case it's appropriate to use `keyboard.nextInt()`. For instance you might have:

```
int eggs, dozens;

System.out.print ("How many eggs do you have? ");
```

```
eggs = keyboard.nextInt();
dozens = eggs / 12;
System.out.println ("You have " + dozens + " dozen eggs.");
```

Note the use of integer division here, producing an integer quotient. Usually it is best to declare a variable as an int if you know that the value will be integral. This avoids questions of precision that arise with doubles. On the other hand, there are many times when you need to use numbers that are not integers.

The only difference between `System.out.print` and `System.out.println` is that `println` always adds a carriage return at the end of the line and `print` doesn't. They are appropriate at different times. If you are asking the user to enter a value, you'll usually use `print`; if you're printing results you'll usually use `println`. The `print` and `println` methods can be used to print simple strings:

```
System.out.println ("Hi there.");
```

They can also be used to print numbers:

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

This causes whatever value is in `myAge` to be printed. The information can be more complex:

```
System.out.println ("You have " + dozens + " dozen eggs.");
```

Whenever the plus sign, `+`, is applied to a string, it means *concatenation*, not addition. Suppose `dozens` contains the number 5. Concatenating "You have " with `dozens` yields "You have 5". Concatenating this with " dozen eggs." yields "You have 5 dozen eggs." It is also possible to do arithmetic computations within a `println`. The previous statement is equivalent to:

```
System.out.println ("You have " + (eggs/12) + " dozen eggs.");
```

2.10 An Example Reading Integers and Strings

Figure 2.4 is another program demonstrating simple input and output. Here's an example of what happens when it is run:

```
What's your name? Nancy
What's your age? 26
Welcome Nancy, I hope you enjoy programming in Java.
You seem to be 26 years old.
```

```
1   import java.util.Scanner;
2
3   public class Greet {
4
5       //  An example reading strings
6       //                          L. McGeoch, 9/2004
7
8       public static Scanner keyboard = new Scanner(System.in);
9
10      public static void main (String[] args) {
11
12          String name;       // the user's name
13          int    age;        // the user's age
14
15          System.out.print ("What's your name? ");
16          name = keyboard.nextLine();
17
18          System.out.print ("What's your age? ");
19          age = keyboard.nextInt();
20
21          System.out.println ("Welcome " + name +
22              ", I hope you enjoy programming in Java.");
23          System.out.println ("You seem to be " + age + " years old.");
24      }
25  }
```

Figure 2.4: A program to do a personal greeting.

This example shows the declaration and use of an new kind of variable, a *string*. A string is similar to an int or a double, but it holds a sequence of characters. You can write

```
String s = "Hello";
s = s + " " + "there";
```

This pair of statements gives `s` the value "Hello" and then modifies it to contain "Hello there". A quoted string is a string constant, in the same way that 3 is an int constant.

We'll see later that `String` is actually the name of a Java class that contains code that allows strings to be handled in sophisticated ways. Class names are always capitalized, and therefore you must always use the capitalized word `String` when you declare a string variable.

The `keyboard.nextLine` method permits a string to be read from the keyboard. It reads characters from the keyboard, up to the next newline character. The newline character, also known as a *carriage return*, *end-of-line mark*, or *line break*, is generated when you type the return key on a keyboard. The method returns a string containing all of the characters that were read, not including the newline. Usually the method that called `keyboard.nextLine` will assign the value to some variable.

Lines 21 and 22 of this program are an example of how a single statement can be spread across multiple lines of a program. The entire call to `println` would be too wide to fit on the page if it were all on one line. By inserting a line break before the last string constant, you can fit the statement on the page without changing its meaning and without causing too much confusion. Java does not permit line breaks within string constants.

2.11 Style

As we discussed in Chapter 1, good programming style makes programs more readable for others and ultimately for you. Examine the programs in this chapter, and throughout the book, carefully and try to imitate the style.

Comments are essential in explaining a program. Each class should have comments at the beginning, explaining what the program does. So far our programs have been simple and have had brief explanations; they'll get more detailed later. Be sure to include your name and the date, to establish the history of the program. Within your program, you should use comments to explain variables. If a program is complex, you can explain how its methods and subparts work together.

There are two ways of delimiting comments, the `//` mark, which indicates that the rest of the line is a comment, and the `/*` mark, which begins a comment that runs until the next `*/`. For example:

```
//   Here's our usual kind of comment.
/*   Here's a comment that keeps
         going,
         and going,
         and going.       */
```

Usually you'll want to use the first kind, because it is less likely to lead to confusion, but occasionally it will be more clear to use the second kind.

You should use *spacing* to make your programs more clear. The computer ignores spacing in Java programs, except in string and char constants. (We'll talk about the type *char* later.) The program in Figure 2.4 uses blank lines between different

subparts, visually breaking up a long sequence of statements. *Indentation* is an especially critical way to indicate the structure of a program. The class `Greet` is the top-level entity in the program; lines 3 and 25, which delimit the class, are not indented at all. The method `main`, which is enclosed in `Greet`, is indented by one tab stop. The statements within `main`, lines 12 through 23, are indented by *two* tab stops. As your programs become more complex, there will be more sublevels with various levels of indentation.

2.12 Debugging

Section 1.6 described the three kinds of errors that might occur in programs: compilation errors, run-time errors, and logical errors. Let's look more closely at what happens when your program has errors, as well as strategies for locating and fixing errors.

Common compilation errors include misspelled identifiers and reserved words, misplaced punctuation, and failure to declare variables. Suppose you leave out a semicolon in a program. When you compile the program in Figure 2.3, you might get the following message:

```
./TempCalc.java:14: ';' expected.
    double kelvin
                  ^
1 error
```

This gives the location of the error (line 14 in `TempCalc.java`), a brief explanation (a semicolon was expected but missing), the actual line with the error, and a mark (the carat symbol, ^), showing where the error is in the line. To fix this program, you simply need to edit the program, insert the missing semicolon, save the modified program, and recompile.

Sometimes error messages are more confusing. For instance, consider the following messages:

```
TempCalc.java:19: Invalid left hand side of assignment.
    celsius = (fahrenheit-32) * (5.0 / 9.0)
                              ^
TempCalc.java:20: Variable celsius may not have
                  been initialized.
    kelvin = celsius + 273.15;
             ^
2 errors
```

In this case, there's a single error: a missing semicolon at the end of line 19. The compiler is confused by the error and considers the two lines to constitute a single meaningless statement. If you get multiple error messages, you may have multiple errors, or you may have just one. If you have difficulty understanding the error messages, concentrate on the first one. If you can't locate an error immediately, look at the lines immediately before or after the one that the compiler claims has the error.

Appendix B describes some of the error messages you might see when you compile a program. Use it to help determine the likely meaning of a message. Appendix A contains a list of common programming errors. Think about these errors, because nearly all programmers have encountered these errors at one time or another.

Run-time errors occur if something goes wrong while a program is running, but after it has been successfully compiled. Suppose you have a program that asks the user for two integers and then divides them and prints the quotient. When you run the program, the following might happen:

```
> java Div
Enter two integers:
3 0
3 divided by 0 is
java.lang.ArithmeticException: / by zero
 at Div.main(Div.java:9)
```

An `ArithmeticException` occurs if you divide an integer by zero, because no such division is possible. This message says that the problem occurred on line 9 of method `main` in file `Div.java`.

Logical errors occur if you write statements that don't do what you expect them to do. Simple examples might include storing a result in the wrong variable (for instance, putting a Fahrenheit temperature in variable `celsius`) or doing integer division when you intended to do floating-point division.

The errors you encounter based on the ideas you learned in this chapter are likely to be fairly straightforward. We'll return to the discussion of debugging and preventing errors later, once you've learned more sophisticated programming techniques.

2.13 Summary

This chapter discussed how you can write, compile, run, and debug simple programs. The key ideas include:

1. *Identifiers:* names chosen by the programmer for methods, variables, and classes.

2. *Reserved words:* words with special meaning that cannot be used as identifiers.

3. *Variables:* places in computer memory that can hold values. Each variable has a particular type. We've discussed three types so far: int, double, and string.

4. *Declarations:* statements that define variables, making them available for use in a program. Some declarations include initialization to assign a value to the variable.

5. *Expressions:* computations that yield values. An expression can be used anywhere a value is expected and can be based on variables, constants, or operators. *Precedence rules* determine the order in which operations should be done in computing a value.

6. *Assignment statements:* statements that take a value and copy it into a variable.

7. *Input and output methods:* methods that read information from the keyboard or write it to the screen.

Programs should use comments and good style, including good spacing and indentation. This can make them much more understandable and much easier to debug.

Chapter 3

Making Decisions

Computer programs can be designed so that different inputs cause different actions to occur. This is illustrated in the main method shown in Figure 3.1.[1]

```
1       public static void main (String[] args) {
2
3           System.out.print ("Enter an integer: ");
4
5           int a = keyboard.nextInt();
6
7           if (a > 0)
8               System.out.println ("Your value is positive.");
9
10          else {
11              System.out.println ("Your value is not positive.");
12              System.out.println ("Too bad!");
13          }
14
15          System.out.println("Goodbye!");
16      }
```

Figure 3.1: A main method using an if-then-else statement

Notice the use of the reserved word `if` on line 7. This is the beginning of an *if-then-else* statement. When the program runs, the test condition `a > 0` on line 7 is executed. If variable `a` is positive, the *then-clause* is executed; if it isn't, the

[1]For brevity, many of our example programs will omit the header material that occurs before the first method, such as the line defining the name of the class and the lines that create the Scanner object. In order to compile a program you will still need to include these lines.

else-clause is executed. In this case the then-clause is the single statement on line 8, and the else-clause is the *block* of statements beginning on line 10. After executing one of the two clauses, the program will go on to line 15.

It is also possible to have an if statement without an else-clause. This is called an *if-then* statement. Here's an example:

```
if (a > 0)
    System.out.println ("a is positive");
if (b < a)
    System.out.println ("b is less than a");
System.out.println ("Now we're done.");
```

In this example, the first test if applied. If `a` is positive, the first `println` statement is executed. Then, whether or not `a` was positive, the second test is applied. Depending on the outcome of the second test, the second `println` might be used. Finally, the third `println` is executed.

We can also put statements together in more complicated ways:

```
if (a > 0) {
    System.out.println ("a is positive");
    if (b > 0)
        System.out.println ("and so is b");
    else
        System.out.println ("but b isn't");
}
else
    System.out.println ("a is not positive");
```

In this case the then-clause of the first if-then-else statement encloses a block of statements that contains another if-then-else.

3.1 The Structure of If Statements

In general, the form of an if-then-else statement is:

`if` (*expression*)
    statement
`else`
    statement

The form of an if-then statement is:

`if` (*expression*)
statement

The test expression can have a complex form that we'll discuss later. Often the test expression is a simple comparison of numeric values. There are six operators for comparing numbers:

`==`	*equals*
`!=`	*not equals*
`>`	*greater-than*
`<`	*less-than*
`>=`	*greater-than-or-equals*
`<=`	*less-than-or-equals*

These operators all have lower precedence than any of the arithmetic operators. All of the following are legitimate test expressions:

```
a > b
a != 3*b
i*j <= 2
```

In either kind of if statement, the test expression must be enclosed in parentheses.

The then-clause is a single statement that follows the test expression. It might be a `println` statement, an assignment statement, or even another if statement. The then-clause can also be a *block* of statements surrounded by braces, `{` and `}`.

Here's another example:

```
if (a != 3*b)
    i = 1;
else
    i = 2;
```

In this case `a` is compared to `3*b`. If they are unequal, the then-clause is executed; otherwise the else-clause is executed. In this case each clause contains a single assignment statement. It's fine to add braces to make a clause a block:

```
if (a != 3*b)
    i = 1;
else {
    i = 2;
}
```

We can then add a second statement to the block:

```
if (a != 3*b)
    i = 1;
else {
    i = 2;
    j = 3;
}
```

Here's an incorrect if-then-else:

```
if (a != 3*b)              // an incorrect example
    i = 1;
    j = 4;
else {
    i = 2;
    j = 3;
}
```

The problem here is that there are no braces around the intended then-clause. The compiler interprets the assignment `i = 1;` as the entire then-clause. It then expects to see either the keyword `else` or some other statement. When it sees the assignment to `j` it decides that an if-then statement has ended. When it later sees the keyword `else` it issues a error message saying that there is an else-clause without a corresponding if statement.

It's important to use spacing and indentation well to indicate your intentions to human readers of your programs. As we discussed earlier, the compiler doesn't care about spacing or indentation at all, but the layout of your program can either hinder or help others who read your programs. Try to imitate the layout I've used in the examples above, indenting the body of the then- and else-clauses one step in from the if statement itself. Here's an example of how indentation can be misused:

```
if (a != 3*b)              // a confusing program
    i = 1;
    j = 4;
k = 3;
```

It's hard to tell what's going on in this example. Does the programmer intend to have the assignment to `j` be part of the then-clause? If so, braces need to be placed around the then-clause. On the other hand, if the assignment to `j` is supposed to follow the if-then statement, it should be lined up underneath the keyword `if`.

3.2 An Example: Finding the Median

The median of three numbers is the "middlemost" of the three values. The median of 4, 2 and 3 is 3. The median of 6, 2, and 2 is 2. Figure 3.2 shows a program that determines the median of three integers given by a user. In this program there are three levels of if statements. Proper use of indentation helps expose the structure of the program, and comments on key lines help explain its logic.

You should usually try to initialize each variable to an appropriate value when it is declared. The median-finding program shows a case in which no initialization is appropriate. There is no reasonable starting value for variable `median`. We simply declare it without initialization and then assign values in the right places.

3.3 An Example: Solving Quadratic Equations

A quadratic equation has the form

$$ax^2 + bx + c \ = \ 0,$$

with $a \neq 0$. In an algebra class you may have learned that this equation is satisfied by any x such that

$$x \ = \ \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

The value $b^2 - 4ac$ is known as the discriminant. If the discriminant equals zero, there is only one real root x that satisfies the equation. If the discriminant is positive, there are two distinct real roots. If the discriminant is negative, there are two distinct complex roots.

Figure 3.3 shows a program that computes the real roots of a quadratic equation. It begins by prompting the user to enter three integers for coefficients a, b, and c. (The program ignores the requirement that a be non-zero.) It computes the discriminant, which is integral. It also computes the value `r`, which might not be integral. To ensure that real division occurs, I've used 2.0 in computing the divisor.

The first if-then-else statement begins on line 24. If the discriminant is zero, the program prints the one value that satisfies the equation. Otherwise it tests for the case in which the discriminant is positive and handles it appropriately. Finally it handles the case in which the discriminant is negative.

There are several things to notice in this code. First, line 28 shows the declaration of a variable *within* a block of statements. The variable exists and is usable only within the block. The same variable name can be declared and used within separate blocks. Line 28 also shows the use of the method `Math.sqrt`. This method

```
1   import java.util.Scanner;
2
3   public class Median {
4
5       // This program finds the median of three integers
6
7       public static Scanner keyboard = new Scanner(System.in);
8
9       public static void main (String[] args) {
10
11          System.out.print ("Enter three integers: ");
12
13          int a = keyboard.nextInt();
14          int b = keyboard.nextInt();
15          int c = keyboard.nextInt();
16          int median;
17
18          if (a < b) {
19              if (b < c) {
20                  median = b;
21              }
22              else {    // b is the biggest
23                  if (a > c)
24                      median = a;
25                  else
26                      median = c;
27              }
28          }
29          else {        // a >= b
30              if (c >= a) {
31                  median = a;
32              }
33              else {    // a is the biggest
34                  if (b > c)
35                      median = b;
36                  else
37                      median = c;
38              }
39          }
40          System.out.println ("The median value is " + median);
41      }
42  }
```

Figure 3.2: Finding the median of three integers

```
1   import java.util.Scanner;
2
3   public class Quadratic {
4
5       // This program solves quadratic equations
6       //                      L. McGeoch, 9/2004
7
8       public static Scanner keyboard = new Scanner(System.in);
9
10      public static void main (String[] args) {
11
12          System.out.println ("This program solves quadratic equations of the");
13          System.out.println ("form a*(x^2) + b*x + c = 0.");
14          System.out.println ();
15          System.out.print ("Enter integers for a, b, and c: ");
16
17          int a = keyboard.nextInt();
18          int b = keyboard.nextInt();
19          int c = keyboard.nextInt();
20
21          int discriminant = b*b - 4*a*c;
22          double r = -b / (2.0 * a);
23
24          if (discriminant == 0) {
25              System.out.println ("There is one root: " + r);
26          }
27          else if (discriminant > 0) {
28              double d = Math.sqrt (discriminant) / (2.0 * a);
29              System.out.println (" There are two real roots: "
30                                  + (r+d) + " and " + (r-d));
31          }
32          else {
33              System.out.println ("The roots are complex numbers.");
34          }
35      }
36  }
```

Figure 3.3: Computing the roots of a quadratic equation

is one of a set of mathematical methods that can used in any Java program. We'll discuss this more fully later.

The layout of the if statements is also a little different than the layout we used earlier. This program follows this general form:

```
if ( expression )
      statement
else if ( expression )
      statement
      ⋮
else if ( expression )
      statement
else
      statement
```

This form is appropriate if you want to do exactly one action chosen from some set of actions. The first test decides if the first action applies. If it fails the second test is applied, and so on. Technically each if statement lies entirely in the else-clause of the previous statement. Instead of indenting the alternatives more and more deeply, we illustrate their parallel nature by lining them up at a single indentation level.

3.4 Boolean Expressions

A *boolean expression* is an expression which has a value of either true or false. The test expressions we used earlier in this chapter, obtained by comparing numbers, are one kind of boolean expression. The boolean expression

```
a < b + c
```

has value true if the value of `a` is less than the sum of `b` and `c`. We can construct more complex boolean expressions, for example:

```
(a < b) && (c > d)
(i >= 0) || !(c > e)
```

These expressions illustrate the use of three new operators: `&& ||`, and `!`. The operator `&&` means *and*. It yields the value true if the values on its left and right are both true. The first boolean expression here is true if both `a < b` and `c > d` are true.

```
1   import java.util.Scanner;
2
3   public class Median2 {
4
5       // This program finds the median of three integers
6
7       public static Scanner keyboard = new Scanner(System.in);
8
9       public static void main (String[] args) {
10
11          System.out.print ("Enter three integers: ");
12
13          int a = keyboard.nextInt();
14          int b = keyboard.nextInt();
15          int c = keyboard.nextInt();
16          int median;
17
18          if ((a <= b && b <= c) || (c <= b && b <= a))
19              median = b;
20          else if ((b <= a && a <= c) || (c <= a && a <= b))
21              median = a;
22          else
23              median = c;
24
25          System.out.println ("The median value is " + median);
26      }
27  }
```

Figure 3.4: Another way to find the median

The operator `!` means *not*. It takes whatever value is on its right and flips it, either from true to false or vice versa. The operator `||` means *or*. It takes the values on its left and right and yields true if either or both values are true. The second expression shown above is true if either `i >= 0` or `c <= e` or both.

In general we will use boolean expressions in ways that are fairly straightforward. For example, see Figure 3.4, which gives a revised version of the median-finding program. Try reading the if statements in this program as if they were English sentences.

The precedence of the operator `&&` is higher than `||`. So in the expression

```
a < b  ||  c < d  &&  e < f
```

the results of the second and third comparisons are "anded" before the "or" operation.

Here's a consolidated list of operators we've discussed so far:

`- !`	*negation* and *not*
`(`*type*`)`	*casting*
`* / %`	*multiplicative operators*
`+ -`	*additive operators*
`< > <= >=`	*relational operators*
`== !=`	*equality operators*
`&&`	*logical and*
`||`	*logical or*

Each line lists operators with equal precedence, and the operators on each line have higher precedence than those on the next. When in doubt, use parentheses to specify the order in which operations should be done. Notice that the minus sign `-` appears twice on the list. Negation, for example in `j = -i`, has very high precedence, while subtraction has lower precedence.

3.5 The Dangling Else Problem

Consider the following code:

```
if ( a < b )
    if ( c < d )
        i = 0;
else
    j = 1;
```

This code could be interpreted two ways. There could be an if-then statement in the then-clause of an if-then-else. That is, it could be equivalent to this code:

```
if ( a < b ) {
    if ( c < d )
        i = 0;
}
else
    j = 1;
```

Alternatively there could be an if-then-else statement in the then-clause of an if-then. That is, the original code could be equivalent to the following:

```
if ( a < b ) {
    if ( c < d )
        i = 0;
    else
        j = 1;
}
```

The existence of two interpretations of the original code fragment is called the *dangling else problem*. The indentation suggests that the programmer intended the first interpretation, but the compiler will use the second. The compiler's rule is: *an else-clause is part of the closest if to which it can be attached.* Nearly every programmer eventually writes a program that is incorrect because it includes code similar to the original fragment. Beware, and be sure to use braces to disambiguate the meaning of your programs.

Chapter 4

Looping

Most interesting computer programs use *loops* to perform repeated computation. Figure 4.1 gives a simple example. In this example we begin by declaring an integer

```
1       public static void main (String[] args) {
2
3           int i=1;
4
5           while (i <= 10) {
6               System.out.println ("Hello " + i);
7               i = i + 1;
8           }
9       }
```

Figure 4.1: A program with a while loop

`i` and initializing it to the value 1. We then use a *while loop* to repeatedly print out a message. The statements inside the loop are executed ten times. During each pass through the loop, a line is printed and the value in `i` is incremented. The effect is to print

```
Hello 1
Hello 2
   ⋮
Hello 10
```

The general form of a while statement is:

`while` (*expression*)
statement

The *expression* can be any boolean expression. The body of the while can be any statement at all, including a block of statements enclosed by braces.

When a while statement is reached, the expression is evaluated. If it is false, the body is skipped and execution resumes at the statement following the while statement. Consider the following code:

```
System.out.println ("Hi");
while (1 < 0)
    System.out.println ("This is never printed");
System.out.println ("Bye");
```

This code first prints the word `hi`. The test expression is false, so the body of the while is skipped and the word `bye` is printed.

If the test expression is true, the body of the while is executed and then the test expression is reevaluated. The "loop" of testing the expression and executing the body may be repeated many times until eventually the test expression becomes false. In the program in figure 4.1, the expression is evaluated eleven times, with `i` equal to 1, 2, ..., 11. When `i` is 11, the expression is false and the body is skipped.

One common programming error is creating an *infinite loop*. Consider the following code:

```
int i = 0;
while ( i < 10 )
    println ("Hi");
```

Notice that the variable `i` is never changed within the loop. The condition `i < 10` will *always* be true and the program will keep running and running and running. Most systems have some way of terminating programs that are stuck in infinite loops. For example, on a Unix system you can usually type `CTRL-C` to end a program.

4.1 A Second Example

Consider the code in Figure 4.2. This program asks the user to enter a sequence of integers, terminated with a zero. The zero isn't considered to be part of the sequence. The program repeated reads a number and tests whether or not it's a zero. If it is, the program ends; otherwise a count is incremented and a new number is read.

```
1       public static void main (String[] args) {
2
3           System.out.println ("Enter a sequence of integers.");
4           System.out.println ("Type a zero to mark the end of the sequence.");
5
6           int count = 0;        // how many numbers have been read?
7           int number = keyboard.nextInt();
8
9           while (number != 0) {
10              count = count + 1;
11              number = keyboard.nextInt();
12          }
13          System.out.println ("There were " + count
14                              + " numbers in the sequence.");
15      }
```

Figure 4.2: Another program with a while loop

(You may ask "Is this a realistic way to input a sequence of integers?" It depends on the situation. Obviously this approach won't work if you need to permit zero as a legitimate element of the sequence. We'll see more elegant ways to handle sequences of data later; for now this simple approach will serve us well.)

4.2 Boolean Variables

In Section 3.4 we introduced the idea of a boolean expression. When they are evaluated, boolean expressions yield either true or false. Java also permits the declaration of *boolean variables*. Consider the following sequence of instructions:

```
boolean i = true;
boolean j = 3>4;
i = i && j;
```

In this sequence, two variables `i` and `j` are declared. Variable `i` initially contains the value true (generated by the keyword `true`) and `j` contains false (which could have been generated by the keyword `false`). In the third statement the two values are "anded," yielding the value false, which is stored in `i`.

boolean is a full-fledged type in Java, just as int and double are types. There are two boolean constants, true and false. Every boolean expression evaluates to either true or false and every boolean variable contains one of these values. Boolean values can only be stored in boolean variables and they can't be used in a place where a

number is expected. Here are some examples of illegal statements that will cause compiler errors:

```
boolean b = 3;       // some bad examples
int i = 3<4;
double d = false;
```

In each case, the value that is being assigned is not compatible with the variable that is the target of the assignment.

Boolean variables are often used in a fairly simple way. For example, consider the following code:

```
boolean done = false;

while (done == false) {
    System.out.print ("Type an integer (0 means quit): ");
    int i = keyboard.nextInt();
    if (i == 0)
        done = true;
    else
        System.out.println ("You typed " + i);
}
```

In this code the variable `done` is used to indicate whether or not the task is done. Initially `done` is false. When the user types the right value, `done` is set to true and the loop ends.

Interestingly, the test condition could have simply been `!done`. The boolean expression `!done` is true if and only if `done` is false. Similarly the boolean expression `ok` is equivalent to the expression `ok == true`. Both are true if and only if `ok` is true.

4.3 A Bigger Example

The code in Figure 4.3 extends the program we considered in the previous chapter. The program maintains a running sum of the numbers in the sequence. It also detects if the sequence is *non-decreasing*, that is, if no value is smaller than the one before it. The sum is maintained by declaring and initializing a variable `sum` on line 8. Each new number is added to the current sum on line 14. The statement on line 14 is equivalent to

```
sum = sum + number;
```

```
1       public static void main (String[] args) {
2
3           System.out.println ("Enter a sequence of integers.");
4           System.out.println ("Type a zero to mark the end of the sequence.");
5
6           int count = 0;        // how many numbers have been read?
7           boolean ok = true;    // is sequence non-decreasing?
8           int sum = 0;          // what's the sum so far?
9
10          int number = keyboard.nextInt();
11
12          while (number != 0) {
13              count++;
14              sum += number;
15
16              int newnumber = keyboard.nextInt();
17              if (newnumber != 0 && newnumber < number) ok = false;
18              number = newnumber;
19          }
20          System.out.println ("There were " + count
21                              + " numbers in the sequence.");
22          System.out.println ("The sum is " + sum + ".");
23
24          if (ok)
25              System.out.println ("Sequence is non-decreasing.");
```

Figure 4.3: A bigger example of a while loop

and the statement on line 13 is equivalent to

```
count = count + 1;
```

We'll talk about these new assignment operators in a bit.

To detect whether or not the sequence is non-decreasing, we introduce a boolean variable `ok`, which is initially true. When we read a number, we place it in a new variable `newnumber`. Each time we read a number, we compare it to the previous one. If it's smaller, we set `ok` to be false. After testing whether or not the sequence is non-decreasing, we copy `newnumber` into `number` so it will be ready for the next comparison.

4.4 Assignment Operators

The following operators permit you to combine arithmetic with assignment:

```
+=   -=   *=   /=   %=
```

Each of them works by computing the value on the right-hand side and by then doing arithmetic to combine it with the variable on the left. For example, suppose you had:

```
i *= i+j+k;
```

This would cause variable `i` to be multiplied by the value on the right-hand side. In other words, it would be equivalent to

```
i = i * (i+j+k);
```

Java also uses two operators, `++` and `--`, for *increment* and *decrement* operations. Writing either `i++` or `++i` is equivalent to `i = i+1`, and writing either `i--` or `--i` is equivalent to `i = i-1`. The placement of the operator either before or after the variable is significant only if the increment or decrement is part of a larger expression. For example, consider the statement

```
j = i++;
```

This would copy the value of `i` into `j` and would then add one to `i`. On the other hand, the statement

```
j = ++i;
```

would add one to `i` before copying the value to `j`. In other words, the placement of the `++` operator before or after the variable indicates whether the increment should take place before or after the variable's value is used.

4.5 For Loops

A *for loop* is a variant of a while loop. Consider the following code:

```
for (i=0; i<n; i++) {
    System.out.println ("Hello");
    System.out.println ("The number is " + i);
}
```

This code is equivalent to the following while loop:

```
i = 0;
while (i < n) {
    System.out.println ("Hello");
    System.out.println ("The number is " + i);
    i++;
}
```

The general form of a for loop is:

`for` (*initialization* ; *expression* ; *increment*)
    statement

It is equivalent to:

initialization ;
`while` (*expression*) `{`
    statement ;
    increment ;
`}`

Figure 4.4 shows another example of a for loop. This code fragment tests the primality of a number (greater than one) entered by the user. The initialization part of the for loop declares and initializes the variable `i`. Each value from 2 to `p-1` is tested as a divisor of `p`. If any value evenly divides `p`, the boolean variable `prime` is set to false.

If a variable is declared in the initialization clause of a for loop, its *scope*, the part of the program in which it can be used, is limited to the for statement. In this example, you could not refer to `i` anywhere after the for statement.

```
1           int p = keyboard.nextInt();
2           boolean prime = true;
3
4           for (int i=2; i<p; i++) {
5               if (p % i == 0) prime = false;
6           }
7
8           if (prime)
9               System.out.println ("The number is prime.");
```

Figure 4.4: Primality testing

4.6 Do-While Loops

Here's an example of another kind of loop:

```
i = 0;
do {
    System.out.println ("Hello");
    i++;
}
while (i < n);
```

The general form is:

```
do
      statement
while ( expression ) ;
```

The only difference between a do-while statement and a while statement is that the test is at the end. This means that the body of a do-while will always be executed at least once. Usually it's appropriate to use a while loop, but we'll see examples later in which it's better to use a do-while.

4.7 Nested Loops

Sometimes it is useful to have loops within loops. (This is called *nesting*, and we say that one loop is *nested* within the other.) Figure 4.5 shows a program that uses nested loops to produce a 10-by-10 multiplication table.

```
1       public static void main (String[] args) {
2
3           for (int i=1; i<=10; ++i) {
4
5               for (int j=1; j<=10; ++j)
6                   System.out.print (i*j + " ");
7
8               System.out.println();
9           }
10      }
```

Figure 4.5: A program that prints a multiplication table

When the multiplication table program is run, it 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
```

This output is a bit ugly, not quite what you'd expect in a multiplication table, but the numbers are all correct. We'll talk about a way to line things up properly in the next chapter.

Let's see how the program works. The two for loops are based on different variables. The outer loop uses variable `i` to keep track of which row is being printed. The inner loop uses variable `j` to keep track of the column. We could have called these variables `row` and `column` respectively.

Each pass through the `i` loop causes one complete row to be printed. Let's consider what happens when `i` is 1. The inner loop runs, causing `j` to take on all values from 1 to 10. For each value of `j`, the number `i*j` is printed, followed by a

space. The value of `i` is 1, so the values printed are 1, 2, 3, ... When the `j` loop ends, the `println` adds a carriage return to the end of the line, yielding the line:

```
1 2 3 4 5 6 7 8 9 10
```

When `i` is 2, the inner loops runs again, causing `j` to take on the values 1 through 10 again. For each value of `j` the number `i*j` is printed. Because `i` is now 2, the values printed are 2, 4, 6, ... After the last pass through the inner loop, the carriage return is added, yielding the complete second line:

```
2 4 6 8 10 12 14 16 18 20
```

The process repeats for each `i` from 1 to 10, and the final result is the multiplication table shown above.

Chapter 5

Using Methods

The Java API (Application Programming Interface) is a set of classes that are available to all Java programmers. In order to run Java programs, your system must have both the Java virtual machine and the entire API. The Java API is essentially a vast library of prewritten code that simplifies programming and helps programmers avoid reimplementation of common code.

This chapter will begin by discussing two classes in the API, `Math` and `String`. Full documentation for the entire API is available on-line at

```
http://download.oracle.com/javase/6/docs/api/
```

Both `Math` and `String` are in a *package* called `java.lang`.

5.1 Static Methods

In Chapter 3, one of the programs used a method called `Math.sqrt` to compute the square root of a number. The documentation for the class `Math` describes the method `sqrt` in the following way:

```
public static double sqrt (double a);
```

Here are some other methods in the class:

```
public static int max (int a, int b);
public static double max (double a, double b);
public static double random();
public static double cos(double a);
```

These are sometimes called the *header lines* for the methods. Each of the methods listed here does a computation more or less related to its name. The `max` methods

determine which of two numbers is larger. The `random` method picks a random real between zero and one. The `cos` method computes a cosine. Figure 5.1 shows examples of how these methods might be used.

```
1           double d = 3 * Math.sqrt(7.0);
2           d = 4 + Math.sqrt(5);
3           System.out.println (Math.cos(d));
4           int randomDie = (int)(6.0*Math.random()) + 1;
5           int i = Math.max (3*j, 17);
6           d = Math.max (3.1, i);
```

Figure 5.1: Using class methods

When you *call* a method, you give it *arguments*, values that it will use in its computation. When the method has done its work, it will return a *result* that can be used in some other expression. The *type* of a method is simply the type of the result it returns.

Let's look at the header line for the method `sqrt`. The word `public` at the beginning means that the method can be accessed in other classes and packages. The word `static` means that this is a *class method*. (We'll talk about the alternative in a bit.) The word `double` before the name `sqrt` means that the method will return a double result. The material within the parentheses is the *parameter list*, which specifies a name and type for each argument that is passed to the method. The type of each parameter is important, but the name doesn't really matter. The parameter list for `sqrt` specifies that the method expects a single argument of type double.

Now look at the example method calls in Figure 5.1. In each case the method name has the prefix `Math.` This indicates that we are using a `static` method in class `Math`. In line 1, the method `sqrt` is given a double, which is the specified parameter type. In line 2, it's given an int. This works because an int argument will be automatically cast to a double if needed. (The reverse isn't true.)

Line 3 illustrates the use of a trigonometric function. Java trig functions are based on radians, not degrees. Line 4 shows the use of the method `random`. This method, which has no parameters, returns a random double between 0 and 1. More precisely, it returns a value x such that $0 \leq x < 1$. The value isn't really random, but a sequence of calls will give a sequence of values that appear unpredictable. The effect of the statement in line 4 is to simulate a roll of a single die. Do you see why this is true?

Lines 5 and 6 illustrate calls to two methods that have the same name. One version of `max` expects two int arguments, which the other expects two double arguments. The call on line 5 has two int parameters, so the former is used. The call

on line 6 uses a double, so the latter is used.

Let's consider two hypothetical methods that demonstrate other possibilities:

```
public static void someMethod (int i);
public static boolean anotherMethod (int i, double d);
```

Suppose these methods are declared in class `A`. Here are examples showing how they might be used:

```
A.someMethod(3);
if (A.anotherMethod(2,3.0)) ...
```

The type of `someMethod` is *void*, meaning that it returns no value at all. Methods of type void are sometimes used to print something to the screen, and we'll see many other uses later. A method that is void can only be invoked as a free-standing statement, never as part of an expression. For example, the line

```
int i = A.someMethod(3);            // bad example
```

is meaningless and illegal. Method `anotherMethod` takes two arguments of different types. The first argument must be an int and the second must be a double. The method returns a boolean. It can be used in any place a boolean expression can be used, such as in the test condition of an if or in an assignment to a boolean variable.

Because methods return values that can be used in larger expressions, it is possible to use a method call within an argument to some other method. For example, here's a way to compute the maximum of three integer variables `a`, `b`, and `c`:

```
int max = Math.max(Math.max(a,b),c);
```

This works by comparing the larger of `a` and `b` with `c`.

5.2 Instance Methods

As mentioned in Chapter 2, `String` is the name of a class. Individual strings are *objects*. In some ways, `String` objects are similar to ordinary values of type int, double, or boolean. For example, a variable of type `String` can be used as an argument to a method, can be printed, and can be copied. On the other hand, there are also striking differences between objects and ordinary values. In this section we'll focus on the way in which methods can be applied to objects.

The class `String` defines many methods, including the following:

```
public int length();
public boolean startsWith(String prefix);
public String toLowerCase();
public boolean equals(String s);
```

Notice that none of these header lines uses the word `static`. That's because these are *instance methods* that need to be applied to particular `String` objects. These methods are different than the class methods in class `Math`, which were not associated with any `Math` object.

Figure 5.2 shows examples of how these methods might be used.

```
1           String s = "hello";
2           String t = "bye now!";
3           int i = s.length();
4           boolean b = t.startsWith("he");
5           t = s.toLowerCase();
6           if (s.equals(t)) ...
```

Figure 5.2: Using instance methods

The statements in lines 1 and 2 declare and initialize two `String` variables. In line 3, the method `length` is *applied* to the given string. The purpose of the method is to count the characters in the string, so in this case it returns the value 5. The method `startsWith` returns a boolean that tells whether or not the string begins with the given prefix. In this case the answer is `false`. The method `toLowerCase` creates a copy of the given string with all letters converted to lower case. In line 5, variable `t` is set to refer to the new `String` object while the original object remains unchanged. The `equals` method permits two `String` objects to be compared.[1] For technical reasons, to be described later, it's generally a mistake to use the `==` operator to compare two `String` objects.

It is possible to apply several methods in sequence. For example, we can write:

```
if (s.toLowerCase().equals(t)) ...
```

This causes the creation of a lower-case copy of `s`, which is then compared to `t`.

[1] Advanced programmers will recognize that the argument used by the `equals` method is not restricted to being a `String` but rather can be any kind of object. We will ignore this issue for now.

```
1       public static void main (String[] args) {
2
3           String s = "This is a test";
4
5           for (int i=0; i<s.length(); i++)
6               System.out.println (s.substring(i));
7
8       }
```

Figure 5.3: An example using instance methods

Figure 5.3 shows another example of the use of the methods in class `String`. The `substring` method has the following header:

```
public String substring (int startPos);
```

It returns a string consisting of all of the characters starting in position `startPos` of the string to which the method is applied. (The first character in the string is in position 0, the second in position 1, and so on.) Because variable `i` increases on each pass through the loop, the method prints shorter and shorter suffices of the original string. Running this program leads to the following output:

```
This is a test
his is a test
is is a test
s is a test
 is a test
is a test
s a test
 a test
a test
 test
test
est
st
t
```

5.3 Formatted Output

```
1       public static void main (String[] args) {
2
3           for (double x=0; x <= 1.5; x+=0.1)
4               System.out.println (x + " " + Math.cos(x) + " " + Math.sin(x));
5
6       }
```

Figure 5.4: An example using mathematical methods

Figure 5.4 shows a method to produce a table in which each line contains three values: a value (in radians), the cosine of the value, and the sine of the value. Running this method produces the following output:

```
0.0 1.0 0.0
0.1 0.9950041652780258 0.09983341664682815
0.2 0.9800665778412416 0.19866933079506122
0.30000000000000004 0.955336489125606 0.2955202066613396
0.4 0.9210609940028851 0.3894183423086505
0.5 0.8775825618903728 0.479425538604203
0.6 0.8253356149096783 0.5646424733950354
0.7 0.7648421872844885 0.644217687237691
0.7999999999999999 0.6967067093471655 0.7173560908995227
0.8999999999999999 0.6216099682706645 0.7833269096274833
0.9999999999999999 0.5403023058681398 0.8414709848078964
1.0999999999999999 0.4535961214255775 0.8912073600614353
1.2 0.3623577544766736 0.9320390859672263
1.3 0.26749882862458735 0.963558185417193
1.4000000000000001 0.16996714290024081 0.9854497299884603
```

The output is truly ugly! The first value on each line should be a multiple of 0.1, but precision errors mean that the values are sometimes a bit off. We'd also like to have three columns of numbers so the results are easier to read. We can get a cleaner output by replacing line 4 with:

```
System.out.printf ("%3.1f  %5.3f  %8.3f\n", x, Math.cos(x), Math.sin(x));
```

The output now becomes:

```
0.0  1.000     0.000
0.1  0.995     0.100
0.2  0.980     0.199
0.3  0.955     0.296
0.4  0.921     0.389
0.5  0.878     0.479
0.6  0.825     0.565
0.7  0.765     0.644
0.8  0.697     0.717
0.9  0.622     0.783
1.0  0.540     0.841
1.1  0.454     0.891
1.2  0.362     0.932
1.3  0.267     0.964
1.4  0.170     0.985
```

The `printf` method has the interesting feature that it can take a varying number of arguments. The first argument is a string giving the format of the output. The code `%3.1f` means that a floating point number should be printed in field of width three, with one digit right of the decimal point. There are two blanks in the format string before the code `%5.3f`, meaning that there should be two blanks in the output, followed by a floating point number in a field of width at most 5. The code `\n` means that a newline character should be inserted at the end. The three numbers that are printed in the `printf` are obtained by using the three arguments that follow the formatting string.

Here are some more examples:

```
1           System.out.printf ("The answer is %d.\n", answer);
2           System.out.printf ("%d plus %d is %d.\n", 2, 2, 4);
3           System.out.printf ("Here's a number: %6.3f, ", Math.PI);
4           System.out.printf ("and here it is in scientific notation: %7e\n", Math.PI);
5           System.out.printf ("Your name is %s, and your age is %d.\n", "Sue", 20);
6           System.out.printf ("\"Help!\" he cried.\n");
7           System.out.printf ("One half equals 50%%.\n");
8           System.out.printf ("%10s  %6d  %6d\n", "Jim", 43, 28);
9           System.out.printf ("%-10s  %-6d  %-6d\n", "Jim", 43, 28);
```

Here is the corresponding output:

```
The answer is 3.
2 plus 2 is 4.
Here's a number:  3.142, and here it is in scientific notation: 3.141593e+00
Your name is Sue, and your age is 20.
"Help!" he cried.
One half equals 50%.
       Jim      43      28
Jim         43      28
```

Let's look at the examples one by one. In line 1, we use the code `%d` to indicate that an int should be printed as an ordinary decimal number. Line 2 gives an example with three ints. Line 3 shows another example with a floating-point number. Note that we can write `Math.PI` to get the value of π. There is no carriage return code at the end of the line, so the next `printf` continues on the same line. The code `%7e` on line 4 causes a floating-point number to be printed in scientific notation. Line 5 demonstrates the use of the code `%s` to print a string. Line 6 shows how you can write `\"` within a string to put a double-quote mark into a string. Line 7 shows how you can write `%%` within the formatting string to print a percent mark. Line 8 shows how you can use field widths with `%d` and `%s`. Line 9 demonstrates the use of a negative field width. Each argument is still printed in a field of a given width, but each is left-justified within the field.

5.4 Another Example with printf

```
1       public static void main (String[] args) {
2
3           for (int i=1; i<=10; ++i) {
4
5               for (int j=1; j<=10; ++j)
6                   System.out.printf ("%5d ", i*j);
7               System.out.println();
8           }
9       }
```

Figure 5.5: Another example using printf

Figure 5.5 shows a revised version of the multiplication table program from page 53. By using `printf` instead of `print` for the individual numbers, we are able to line up the columns and produce 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
```

Chapter 6

Writing Static Methods

Let's examine how a programmer can write methods to handle subparts of some larger task. Consider the following program, which continues to the next page:

```
1   import java.util.Scanner;
2
3   public class Prime2 {
4
5       ////////////////////////////////////////////////////////
6       // This program does primality testing.
7       //                                  L. McGeoch, 9/2004
8       ////////////////////////////////////////////////////////
9
10      public static Scanner keyboard = new Scanner(System.in);
11
12      ////////////////////// main method /////////////////////////
13
14      public static void main (String[] args) {
15
16          int p = getNumber();
17
18          boolean isPrime = primetest (p);
19          if (isPrime)
20              System.out.println ("It's prime.");
21          else
22              System.out.println ("It's composite.");
23
24      } // main
25
26
27
```

```
28      ////////////////////  getNumber  ////////////////////////
29      //
30      //  This method gets an integer > 1 from the user.
31      //
32      ////////////////////////////////////////////////////////
33
34      private static int getNumber() {
35
36          System.out.print ("Enter an integer greater than one: ");
37          int p = keyboard.nextInt();
38
39          while (p < 2) {
40              System.out.println ("That integer isn't greater than one.");
41              System.out.print   ("Try again: ");
42              p = keyboard.nextInt();
43          }
44          return p;
45
46      } // getNumber
47
48      ////////////////////  primetest  ////////////////////////
49      //
50      //  This method tests whether the given integer is prime.
51      //  It assumes that the parameter is greater than one.
52      //
53      ////////////////////////////////////////////////////////
54
55      private static boolean primetest (int a) {
56
57          int limit = (int) Math.sqrt(a);
58
59          for (int i=2; i<=limit; ++i) {
60              if (a % i == 0) return false;
61          }
62          return true;          // no factors were found
63
64      } // primetest
65  } // class
```

The main method of this program is fairly simple. It invokes two new methods, `getNumber` and `primetest`. The purpose of `getNumber` is to read an integer greater than one from the keyboard, and the purpose of `primetest` is to test the number

for primality.

In this program, code for the two new methods is placed in the same class as the main method. The new methods are *used* by the main method, but they are not *contained* in the main method. Each of the three methods is indented by the same amount, and the closing brace of each method appears before the header line of the next method.

Here's what happens when the program runs. Method `main` begins by calling `getNumber`. The execution of `main` is suspended until `getNumber` completes its work. Method `getNumber` asks the user to enter an integer and saves the value in a *local variable* called `p`. After verifying that the number is greater than one, `getNumber` returns the value to the calling method. Method `main` then stores the value in its own variable `p`.

The main method now invokes method `primetest`, passing the value in `p` as an argument. Within `primetest`, this value is available as a *parameter* called `a`. After using a loop to test possible factors of `a`, `primetest` returns a boolean value indicating whether `a` was prime. Method `main` stores this value in a variable and then uses it to print a message that tells whether the user's number was prime.

6.1 Local Variables, Parameters, and Return Values

Each method has a set of variables that are completely separate from the variables used in other methods. For example, `main` has two variables (`p` and `isPrime`), `getNumber` has one variable (also called `p`), and `primetest` has three variables (`a`, `limit`, and `i`). The variables of each method are *local* to that method and cannot be used in the others.

A method receives information from the rest of the program via *parameters*, and it returns information via a *return value*. The names and types of the parameters and the return type of the method are all found in the header line of the method. As discussed in Chapter 5, the header line

```
private static boolean primetest (int a)
```

tells us that `primetest` takes a single int parameter and returns a boolean value. The keyword `private` specifies that this method can only be called by other methods in the same class. We'll discuss the distinction between public and private methods more later. Your main method must always be public; other methods should be private unless there is a reason to make them public.

Parameters are simply local variables that are given initial values when the method is invoked, that is, when it is called. Suppose a program contains the line:

```
boolean b = primetest(15);
```

When method `primetest` begins, its parameter `a` will contain the number 15. If a method contains the line

```
b = primetest(3*j);
```

then that method's variable `j` will be multiplied by 3 and that value will be placed in parameter `a` of `primetest`.

To return a value, a method must use a return statement. The general form of a return statement is:

`return` *expression* ;

The type of the expression must match the type of the method. The statement `return p;` works in method `getNumber` because an integer return value is expected. The statements `return true;` and `return false;` work in `primetest` because a boolean return value is expected. Unless a method has return type void, it must end with an appropriate return statement.[1] It can also have return statements in other places. For example, `primetest` contains the statement `return false;` in the middle of the loop. If an early return statement of this kind is reached, the method ends immediately and the appropriate return value is passed back to the calling method.

If a method has type void, it has no return value, and no return statement is needed at the end. Here's an example of a void method:

```
private static void sayHi (boolean dontBother) {
    if (dontBother) return;
    System.out.println ("Hi there!");
}
```

While this example is silly, it illustrates several ideas. First, there is no return at the end. If `dontBother` is false, the message is printed and execution "falls out the bottom" of the method. The method terminates, and the calling method resumes. On the other hand, if `dontBother` is true, the return statement, without return value, is executed and the method terminates immediately.

Beginning programmers often write methods that inappropriately use input and output (with the keyboard and display) instead of parameters. Sometimes a student will write a primality testing method that takes a parameter, runs the primality test, and then types a message telling whether or not the number is prime. That's generally the wrong approach; after testing primality, the method should simply return a boolean result. The calling method can either print an appropriate message or can simply use the answer to solve some larger problem.

[1] This rule doesn't hold if for some reason it's impossible for execution to reach the end of the method.

```
1       private static int max (int a, int b, int c) {
2
3           if (a > b) {
4               if (a > c) return a;
5               else return c;
6           }
7           else if (b > c) return b;
8           else return c;
9       }
```

Figure 6.1: Finding the maximum of three integers.

6.2 Why Write Methods?

Methods are useful for a number of reasons. One important reason is that you can write a method one time but use it at many places in a program. For example, you might need to test the primality of an integer `i` at one place in a program and the primality of `j` somewhere else. If you create method `primetest`, you can simply call `primetest(i)` and `primetest(j)` at the appropriate places.

Even if you are only going to do some subtask one time, it often makes sense to create a method. The fact that each method has its own set of variables is crucial. In developing method `primetest`, the programmer doesn't need to worry about where the method is used or what variables the calling method has. He or she can simply concentrate on the fact that the number to be tested arrives in an int variable called `a` and that a boolean needs to be returned. New variables can be declared and used without interfering with the rest of the program. This greatly simplifies the task of programming.

Figure 6.1 shows a method that finds the largest of three integers. This code could be placed into any program and would always work correctly because it only uses local variables. You could use this method by writing

```
int k = max (i+j, i, 4);
```

or

```
k = max (i, j, k);
```

6.3 Another Example of Parameter Passing

Figure 6.2 illustrates a more complex combination of method calls. Each method has its own set of local variables and parameters. Notice that all of the methods

```
1   public class Fun {
2
3       // A silly class demonstrating how parameters and
4       // return values are used.
5       //                                  L. McGeoch, 9/2001
6
7       ////////////////// main method ////////////////////////
8       public static void main (String[] args) {
9
10          int i=5;
11          b();
12          System.out.println ("The answer is " + a(a(i,3.4),3.0));
13
14      }
15
16      ////////////////// method a /////////////////////////////
17      private static int a (int i, double j) {
18
19          System.out.println ("First parameter is " + i);
20          i++;
21          if (c(j)) b();
22          return i;
23      }
24
25      ////////////////// method b /////////////////////////////
26      private static void b () {
27
28          for (int i=0; i<5; ++i)
29              System.out.println ("Hi " + i);
30      }
31
32      ////////////////// method c /////////////////////////////
33      private static boolean c (double i) {
34
35          int j = (int) i;
36          if (i == j) return true;
37          else return false;
38      }
39  }
```

Figure 6.2: Another example with methods and parameters.

have a variable called `i`. These are all distinct variables; changes to one do not affect the others. (By the way, it's unusual and confusing to declare `i` as a double. Variable names `i`, `j`, ..., `n` are generally used for simple integers.)

Let's see what happens in this program. The main method begins. It sets its own variable `i` equal to 5 and then calls `b`. Method `b` prints a message five times and returns. Note that `b`'s use of its own variable `i` doesn't affect the variable `i` that belongs to `main`.

Method `main` contains a pair of nested calls to method `a`. The inner call, with parameter 3.4, occurs first. Once it has ended and returned a value, the outer call is executed.

During the first call to `a`, its variables `i` and `j` receive the values 5 and 3.4 respectively. A message is printed, and `a`'s variable `i` is incremented. The call to `c` returns false (because 3 doesn't equal 3.4), so no call is made to `b`. Finally the current value of `i`, which is 6, is returned back to `main`.

The return value from the first call to `a`, the inner one, is now used as the first parameter of the outer call. On this call to `a`, `i` and `j` begin with the values 6 and 3.0. Variable `i` is again incremented. This time `c` returns true (because 3 does equal 3.0), so method `b` runs again.

The overall output of this program is:

```
Hi 0
Hi 1
Hi 2
Hi 3
Hi 4
First parameter is 5
First parameter is 6
Hi 0
Hi 1
Hi 2
Hi 3
Hi 4
The answer is 7
```

Chapter 7

Declarations, Scopes, and Initializations

In this chapter we'll consider more carefully how declarations work in a program. Let's begin by considering the keyword `int` and how you might use it in a program. Based on the material you've seen so far, there are five possibilities:

1. *Declaring local variables with or without initialization:*

```
int i = 10000;
int j, k;
int m = 2, n = 3;
```

2. *Declaring a parameter:*

```
private static void f (int n) {
```

3. *Declaring the return type of a method:*

```
private static int getNumber() {
```

4. *Declaring a variable in the initialization part of a for statement:*

```
for (int i=0; i<limit; ++i)
```

5. *Casting a value of another type:*

```
int i = (int)(d + 3.0);
```

Four of the uses of the keyword `int` are declarations. You can use the same kind of declarations for other types, including double, boolean, and even `String`.[1]

It's important to make a declaration only when you want to create a new variable. It's tempting to write

```
int i = 3;              // this declaration is fine...
int i = i+1;            // int should be omitted here
```

or

```
if (int i == 3)         // another bad example
```

Be careful to avoid making a declaration when you simply mean to use a variable.

7.1 Scopes

The *scope* of a variable declaration is the section of the program in which the variable can be used.

The scope of a local variable declaration begins at the next statement and continues to the end of the block of statements containing the declaration. For example, consider the following if statement:

```
if (3 < 4) {
    int j = 3;
    int i = j;
    ...
    i = f(3);   // a method call
    int k = 2;
    System.out.println (i*j*k);
}
```

Variables `k`, `j` and `i` can each be used after their respective declarations, but they can't be used outside the then-block. It's important to remember that none of these variables can be accessed in method `f`. Java uses *static scoping*, meaning that the scope of a declaration can be determined by its location in the program, not by the order of method calls. Some programming languages use a different idea, called *dynamic scoping*, in which a method's local variables *are* available in the methods it calls.

The scopes of other kinds of variable declarations are just what you'd expect: the scope of a parameter declaration extends through the entire method containing

[1] More generally, you can use these declarations for objects based on any type or class.

the parameter, while the scope of a declaration in the initialization of a for loop is the entire loop.

Method declarations have scopes too. A method can be used anywhere in the class in which its declared, even in methods that precede it. Methods can be used in other classes, depending on the privacy level (such as public or private) that is assigned to the method. We'll talk about this later.

7.2 Conflicting Declarations of Local Variables

You are not permitted to declare two local variables with the same name and overlapping scopes. The compiler will generate an error for this code:

```
int i=3;

while (...) {
    int i=1;
    ...
}
```

The first declaration of `i` is still in effect within the while loop, yielding overlapping scopes.

The following code is fine:

```
if (...) {
    int i=2;
    ...
}
else {
    int i=3;
    ...
}
```

In this case the scope of the first declaration ends before the scope of the second one begins.

Parameters are treated the same way. If you have a parameter with a particular name, that method can't have a local variable (or another parameter) with the same name.

7.3 Initialization

You should initialize a variable at the time it's declared if there is some reasonable value to assign to it. For example, in a program adding up a sequence of numbers,

it's appropriate to write:

```
int sum = 0;
```

Zero is the natural starting value for this variable. On the other hand, it's best to avoid initialization if there is no good starting value. Consider the following code:

```
int i;

if (...)
    i = f();          // calling method f
else
    i = g();          // calling method g
println (i);
```

The value you want to assign to `i` depends on the result of the if, and there is no sensible initialization.

The compiler will try to detect situations in which you try to use a variable that is uninitialized or potentially uninitialized. For example, the following code will cause an error during compilation:

```
int i;

if (...)
    i = 1;
else if (...)
    i = 2;
System.out.println (i);          // i might be uninitialized
```

If neither of the test conditions yields the value true, variable `i` will not be initialized before the `println` statement. The compiler will determine that `i` might be uninitialized when it is used and will produce an error message. The programmer has probably made a logical error and should have avoided using the second if test. Removing the second if, leaving `i = 2` as the entire else-clause, will eliminate the compilation error.

This example also illustrates the value of avoiding unnecessary initializations. Suppose the programmer had carelessly initialized `i` to zero when it was declared. The compiler would not have complained that `i` might have been uninitialized, and the logical error might not have been detected.

7.4 More Examples with Scopes and Initialization

The following code is erroneous:

```
while (...) {              // i is used outside while loop
    int i = 2;
    ...
}
System.out.println (i);
```

This code is also incorrect:

```
int i;                     // another bad example
while (...) {
    i = 2;
    ....
}
System.out.println (i);
```

In this case the problem is that the body of the while loop might never run. This would occur if the test condition of the while loop failed the first time. If the body is never executed, variable `i` might be uninitialized when it is used in the last statement.

This code works:

```
int i;                     // a good example
do {
    i = 2;
    ...
} while (...);
System.out.println (i);
```

The body of a do loop is always executed at least once, so variable `i` will always be initialized before the `println`.

The following code is incorrect:

```
private static int f () {    // an incorrect method
    ...
    if (...)
        return 1;
    else if (...)
        return 2;
}
```

There are no local variables here, but the return value of a method is essentially a special variable that must be initialized when the method ends. If both of the test conditions fail, execution will fall out the bottom of the method without returning a value. The compiler will detect this situation and produce an error message.

Chapter 8

More Fundamentals

8.1 The Switch Statement

A *switch statement* permits a value to be used to select the code that should be executed. Here's an example:

```
public static void main (String[] args) {

    System.out.print ("Enter an integer: ");
    int i = keyboard.nextInt();

    switch (i) {

    case 2:
        System.out.println ("You entered a two.");
        break;

    case 1:
        System.out.println ("You entered a one.");

    case 4:
    case 3:
        System.out.println ("You entered a 1, 3, or 4.");
        break;

    default:
        System.out.println ("I don't recognize that number!");
```

```
        }
        System.out.println ("Goodbye!");
    }
```

In this case the parentheses surround an expression of type int. Depending on the value of the expression, control jumps to one of the cases. Some number of cases can be explicitly listed, along with a default. If there is a default, the default is used for any value not covered by one of the cases.

The *break statement* causes execution to jump out of the switch. In other words, it causes control to jump to whatever statement follows the entire switch. In the absence of a break, control simply falls through from one case to the next. That's why cases 3 and 4 lead to the same code in the example above.

If there is no default, and if the expression value doesn't correspond to one of the case values, control simply jumps to whatever statement follows the switch. Consider the following:

```
int i = 4;

switch (i) {

case 1:
    System.out.println ("Hi");
    break;

case 2:
    System.out.println ("Bye");
    break;

case 3:
    System.out.println ("Goodnight");
}

System.out.println ("Here we are after the switch");
```

With `i` equaling 4, no case is used and control jumps to the final `println`. If `i` equals 1, 2, or 3, one of the other messages is also printed.

Beware of problems with uncovered cases in a switch. Consider the following:

```
private String monthName (int month) {        // bad example

    switch (month) {
    case 1:
        return "January";
    case 2:
        return "February";
            .
            .
            .
    case 12:
        return "December";
    }
}
```

The compiler will reject this code because it is possible to reach the end without returning a value. You could replace `case 12` with `default` to get a working method.

Similarly the following doesn't work:

```
private String monthName (int month) {        // bad example

    String name;

    switch (month) {
    case 1:
        name = "January";
        break;
    case 2:
        name = "February";
        break;
            .
            .
            .
    case 12:
        name = "December";
    }
    return name;
}
```

This fails because name may be uninitialized at the time of the return. Again, changing `case 12` to `default` is one way to remedy this problem.

Switch statements work with int expressions, boolean expressions, and char expressions, but not with double. (We'll learn about the `char` type in section 8.3.) It's silly to do a switch on a boolean, because the cases are simply true or false.

8.2 Break and Continue Statements in Loops

A break statement can also be used in a loop:

```
private static int getANumber() {

    int sum = 0;

    while (true) {
        System.out.println ("Enter an integer: ");
        int n = keyboard.nextInt();

        if (n < 0) break;
        sum += n;
    }

    System.out.println ("The sum is " + sum);
    return sum;
}
```

If variable `n` is negative, execution will break out of the loop and will continue at the statement immediately following the loop. A break statement always causes execution to jump out of the closest containing switch or loop.

A *continue statement* is similar. Here's an example:

```
for (int n=2; n<1000; ++n) {

    if (isPrime(n)) continue;

    .          // some complex computation occurs here
    .
    .
}
```

For a given value of `n`, the `isPrime` test occurs first. If the value is prime, execution jumps up to the top of the loop and continues with the next value of `n`. So the "complex computation" occurs only for numbers that are not prime. The continue statement is used much less frequently than the break statement.

8.3 Characters

Java uses the type *char* to represent single characters. The following are legal assignments:

```
char c = 'a';
char d = c;
d = ' ';
c = '3';
```

The first of these examples declares a char variable called `c` that contains the letter *a*. The second copies variable `c` into `d`, so `d` also contains the letter *a*. The third fills `d` with a blank, while the fourth puts the digit 3 into `c`. Note that char constants use single-quote marks, not the double-quote marks used in strings.

Figure 8.1 illustrates the use of chars. The method `charAt` is declared in class `String` as

```
public char charAt(int pos);
```

It returns the character in the designated position of the given string. Note that the purpose of the for loop is to go through the characters of the string. If the string has length l, the first character is said to be in position 0 and the last is said to be in position $l - 1$. You will cause an exception if you ask for the character in a position that is negative or greater than $l - 1$.

Class `Character` contains a static method called `toLowerCase`, which has the following header:

```
public static char toLowerCase (char c);
```

If `c` is a letter, it is converted to lower case and returned. If it is a digit, punctuation, space, or anything else, it is simply returned.

Switch statements can be used with chars.

Characters that happen to be digits are not the same as integers. You generally don't want to write

```
sum = sum + c;
```

if `sum` is an integer and `c` is a character.

```
public static int countVowels (String s) {

    int count=0;
    for (int i=0; i<s.length(); ++i) {
        char c = s.charAt(i);
        c = Character.toLowerCase(c);
        switch (c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            count++;
        }
    }
    return count;
}
```

Figure 8.1: An example using chars

8.4 How Characters are Represented

All characters are represented by integers between 0 and 65535. Here are some examples of characters and their *Unicode* representations:

```
' '        32
'!'        33
'"'        34

'0'        48
'1'        49
'2'        50
 .          .
 .          .
 .          .
'9'        57

'A'        65
'B'        66
'C'        67
 .          .
```

```
 .          .
 .          .


'a'        97
'b'        98
'c'        99
 .          .
 .          .
 .          .
```

You can write:

```
if (c >= 'a' && c <= 'z')
```

This essentially asks "Does `c` contain a lower-case letter?" and it works by comparing the character code in `c` to the character codes for *a* and *z*. Another way to ask the same question is

```
if (Character.isLowerCase(c))
```

You can ask if `c` contains a digit by writing

```
if (c >= '0' && c <= '9')
```

or

```
if (Character.isDigit(c))
```

You can convert an upper-case character to a lower-case one with

```
c = (char)(c - 'A' + 'a');
```

When arithmetic is done on a char, it is first converted to the int that represents it in the Unicode system. Subtracting `'A'` from a variable `c` essentially asks "How far into the upper-case letters is the character in `c`?" Adding `'a'` then yields the int that is the same distance into the sequence of lower-case alphabetic character code. The cast to char is needed because char is a special kind of int with a more limited range of values. By using the cast, the programmer acknowledges that he or she understands the special nature of the assignment and expects the value to be in the correct range, 0 through 66535. If the value is out-of-range, the cast yields a character based on the bottom 16 bits of the int.

As noted earlier, the conversion to lower-case can also be done with

```
c = Character.toLowerCase(c);
```

This form would always be preferred because its meaning is more obvious and because it works on all characters, not just those that are upper-case letters. (Characters that aren't upper-case characters are left unchanged.) To convert an entire string, it's even better to use the `toLowerCase` method in class `String`.

8.5 Special Characters

Certain "special characters" have been assigned codes in the Unicode character set. Here are some examples:

`'\n'`	the newline character
`'\t'`	the tab character
`'\0'`	the zero character
`'\''`	the single-quote character
`'\"'`	the double-quote character
`'\\'`	the backslash character

The last three of these aren't really special characters, but special care does have to be taken in order to encode these characters in a character constant or in a string constant. The zero character is rarely used in Java and is different than the digit `'0'`.

Newline and tab characters are often used in building strings that will be printed. For example, one might write

```
System.out.println ("Here's a person:\t" + name + '\t' + age + "\n\n");
```

Each tab character causes the printed string to include one or more blanks, enough to move the output position to the next tab stop. On many systems, tab stops occur every eight characters on the screen or in printed output. The two newline characters mean that the call to `println` prints three lines of output rather than the usual one. The first line contains the information on the person, and the other two are blank.

It would be better and safer to use `"\t"` instead of `'\t'` in the line above. If you wrote

```
System.out.println (age + '\t' + name);
```

the age and tab characters would be added as ints, which is probably not what the programmer intended. Writing

```
System.out.println (age + "\t" + name);
```

will cause both uses of the plus operator to be interpreted as concatenations.

8.6 The ?: Operator

Java has a operator that essentially allows us to embed an if-then-else within an expression. Here's an example:

```
i = (j > k) ? 3 : 4;
```

The `?:` operator involves three parts: a boolean expression that precedes the `?`, a "then value" that comes between the `?` and the `:`, and an "else value" that comes after the `:`. The boolean expression is evaluated, and then either the "then value" or the "else value" is used as the overall value of the expression.

The parentheses aren't actually required in the expression above, because `?:` has lower precedence than any of the operators shown in the chart on page 42. In other words, `?:` could be added as an extra line on that chart. It is often helpful, of course, to use parentheses even when not required, so that an expression's meaning is completely clear to readers (and to you!).

8.7 Assignments are Expressions

Although we have treated assignments as statements, they are actually expressions. Each of these operators

```
=  +=   -=   *=   /=   %=
```

does an assignment that can be part of a larger expression. Here's an example:

```
if ((i = keyboard.nextInt()) > 0) ...
```

In this case, an assignment is made into variable i, and then that value is compared to 0. The rule is: *whatever value is assigned in an assignment is also available for use in a larger expression.*

Here's another example:

```
i = j = k;
```

A sequence of assignments is done, from right to left. The value of `k` is assigned to `j`, and then that same value is assigned to `i`.

The precedence of assignment operators is lower than all of the operators shown on page 42 and than the `?:` operator introduced above. The assignment operators could be given a line of their own at the very bottom of the precedence chart. The precedence rules imply that it is usually necessary to surround an assignment with parentheses when it is part of a larger expression, for example in the if statement above.

The fact that an assignment is an expression creates the possibility of an interesting logical error. If `a` and `b` are boolean variables,

```
if (a == b) ...
```

and

```
if (a = b) ...
```

are legal statements that do very different things. The first simply compares `a` and `b`, while the second copies `b` to `a` and uses that value as the test condition.

Chapter 9

Arrays

An *array* is a set of variables that is created at one time and that can be referenced using a common name. Here's an example:

```
int[] a = new int[100];
```

This declares a variable called `a`. Its type, `int[]`, means "array of ints". The expression `new int[100]` allocates enough memory to hold 100 ints. If variable `a` is declared and initialized in this way, we say that "`a` refers to an array of 100 ints."

It is possible to "index into an array" to access individual elements. Here are some examples:

```
int j = a[50];
int k = 3;
j = a[k];
int m = a[3*a[2]];
a[k] = a[k-1];
```

`a[50]` refers to the 50^{th} integer in the array. `a[k]` uses use whatever value is in variable `k` as the index into array `a`. In this case `k` is 3, so `a[k]` refers to the third integer in the array. Any integer expression can serve as the index. For example, in the third line the value in `a[2]` is multiplied by three, and that value is used to pick an element out of the array.

If an array has n elements, the indices of the element will be from 0 to $n - 1$. The first element of the array always has index 0. An attempt to use a negative index or an index greater than or equal to the array length will cause an `ArrayIndexOutOfBoundsException`.

You can also have arrays of chars, arrays of doubles, arrays of booleans, and arrays of objects. In all cases the indices used are still ints. You can even have arrays of arrays of ints, and so forth. We'll talk about these later.

9.1 Examples of Arrays

Here's a code fragment that generates the first 100 Fibonacci numbers and stores them in an array:

```
int[] fib = new int[100];

fib[0] = 1;
fib[1] = 1;
for (int i=2; i<100; ++i)
    fib[i] = fib[i-1] + fib[i-2];
```

Here's an example in which a loop is used to find the largest value in an array of ints:

```
int[] a = new int[...];     // some length is used

// Values are somehow assigned to elements of the array
...

// Now, we look for the maximum value in the array
int big = a[0];
for (int i=1; i<a.length; ++i)
    if (a[i] > big) big = a[i];
```

Note that `a.length` is a way of asking what the declared length of the array is. The upper limit of the loop is set at `a.length-1` because that's the highest legal index.

Here's an example with strings:

```
String s = "hello";
char[] a = s.toCharArray();
printBackward(a);
```

The method `toCharArray` in class `String` takes the given String and returns an array of chars containing all of its characters. Method `printBackward` is shown in Figure 9.1. Note the use of `char[]` as a parameter type as well as the backward iteration across the array.

Figure 9.1 also contains methods illustrating other ways of using arrays. Method `reverseArray` takes an array of characters and returns a new array that contains the original sequence of characters reversed. Method `readInts` returns an array that is filled based on positive ints read from the keyboard. At the end, `n` contains the number of integers were read, i.e. how much of the array was actually used. Of

```
public static void printBackward(char[] a) {

    for (int i=a.length-1; i>=0; --i)
        System.out.print (a[i]);
    System.out.println();
}

////////////////////////////////////////////////
public static char[] reverseArray (char[] a) {
    char[] b = new char[a.length];

    for (int i=0; i<a.length; ++i)
        b[i] = a[a.length-1-i];

    return b;
}

////////////////////////////////////////////////
public static int[] readInts(int limit) {

    // reads positive ints from the keyboard, up to the given limit

    int[] a = new int[limit];
    System.out.println ("Enter a sequence of positive integers.  Type ");
    System.out.println ("  something non-positive to end the sequence: ");

    int n=0;            // how many have been read

    while (n < limit) {
        int i = keyboard.nextInt();
        if (i <= 0) break;
        a[n] = i;
        n++;
    }
    return a;
}
```

Figure 9.1: Examples using arrays

```
public static boolean paltest(String s) {

    // Returns true if s is a palindrome.
    // Ignores non-letters and is case-insensitive.

    char[] c = s.toLowerCase().toCharArray();
    int i = 0;
    int j = c.length-1;

    while (i < j) {

        if (!Character.isLetter(c[i])) {
            i++;
        }
        else if (!Character.isLetter(c[j])) {
            j--;
        }
        else if (c[i] != c[j])
            return false;
        else {
            i++;
            j--;
        }
    }
    return true;
}
```

Figure 9.2: A method for palindrome testing

course, `n` is a local variable that, in this example, is not returned by the method. This raises the question of how the calling method can determine how much data was read. In this case, it's easy, because arrays of ints are always initialized so that every element is zero. Since zero isn't a positive value, we can tell which part of the array was filled in by the method.

We'll discuss array initialization in section 9.4, and later we will consider more sophisticated ways of keeping track of data when the amount of data is not known in advance.

Figure 9.2 shows a method for testing whether or not a string is a palindrome, i.e. whether or not it reads the same both backward and forward. The method disregards nonalphabetic characters, and it makes no distinction between upper- and lower-case letters. This example uses its loop in an interesting way. Variables `i` and `j` keep track of current positions at the low and high ends of the array. If

the character in position `i` is not a letter, `i` is increased and the loop test is applied again. Similarly, if the character in position `j` is not a letter, `j` is decreased. If there are letters in both positions and if they don't match, the method returns false. If there are letters that do match, `i` and `j` both move towards each other. The loop continues until the two index variables meet. If they do meet, the string must be a palindrome, and the method returns true.

9.2 Arrays of Objects

It's often useful to create arrays of objects. Here's an example using strings:

```
String[] a = new String[10];

for (int i=0; i<10; ++i) {
    a[i] = keyboard.nextLine();
}

for (int i=0; i<10; ++i)
    if (a[i].equals("Hello"));
        System.out.println ("Found it!");
```

This code creates an array that can hold ten strings and fills it with material that is read from the keyboard. It then uses a loop to search through the array looking for the string "Hello".

The use of arrays that hold objects is really identical to the use of arrays that hold primitive types. If `a` is declared as an array of x and if `e` is an integer expression, then `a[e]` is a valid expression of type x. In addition, any value of type x can be stored in `a[e]` in an assignment.

9.3 Matrices

A *matrix* is an array with two or more dimensions. They are needed in many applications and are easy to handle in Java. Figure 9.3 contains code that 1) creates a two-dimensional matrix to hold a multiplication table, and 2) prints the table. Figure 9.4 shows the output that is obtained by running this program.

Let's consider the code in Figure 9.3. Line 2 shows the declaration and initialization of a two-dimensional array. Line 6 illustrates how a new array is created, and line 11 shows the assignment of a value into the array. The first dimension of a two-dimensional array is often referred to as the *row* and the second as the *column*, so variable `i` is the row and `j` is the column. Lines 10 through 12 use an outer

```
1       public static void main (String[] args) {
2           int[][] b = makeTable(10);
3           printTable(b);
4       }
5
6       static int[][] makeTable(int size) {
7
8           int[][] a = new int[size][size];
9           for (int i=0; i<size; ++i)
10              for (int j=0; j<size; ++j)
11                  a[i][j] = i*j;
12          return a;
13      }
14
15      static void printTable(int[][] a) {
16
17          for (int i=0; i<a.length; ++i) {
18              for (int j=0; j<a[i].length; ++j)
19                  System.out.printf("%4d ", a[i][j]);
20              System.out.println();
21          }
22      }
```

Figure 9.3: Creating and printing a multiplication table

```
0    0    0    0    0    0    0    0    0    0
0    1    2    3    4    5    6    7    8    9
0    2    4    6    8   10   12   14   16   18
0    3    6    9   12   15   18   21   24   27
0    4    8   12   16   20   24   28   32   36
0    5   10   15   20   25   30   35   40   45
0    6   12   18   24   30   36   42   48   54
0    7   14   21   28   35   42   49   56   63
0    8   16   24   32   40   48   56   64   72
0    9   18   27   36   45   54   63   72   81
```

Figure 9.4: Output for program in Figure 9.3

loop to iterate over rows and an inner loop to iterate over columns, so the whole first row is filled in before the second row, etc. This is the order that is typically used is processing the elements of a matrix, but there are sometimes good reasons to organize things differently.

Lines 15 through 22 give the code for printing the matrix. The notation `a.length` on line 17 means the length of the first dimension, i.e. the number of rows. The notation `a[i].length` on line 18 means the length of row `i`, i.e. the number of columns.

These two uses of the keyword `length` offer a hint about the true nature of a two-dimensional matrix, which is that it is simply an array of arrays. Each row is an array, and the overall matrix is an array of rows. We'll discuss this idea more fully in Chapter 13.

9.4 Array Initialization

When an array is created, all of the elements are given initial values, depending on the *basetype*, the declared type of the element.

In an array of ...	the initial value of each element is ...
`int`	0
`double`	0.0
`boolean`	`false`
`char`	`'\0'` (the zero character)
objects	`null`
arrays	`null`

`null` is a constant that represents a missing object or a missing array. Suppose you wrote

```
String[] a = new String[10];
```

Variable `a[3]` is initially *null*, meaning that it contains no string at all. It's possible to test if a variable is null by writing something like

```
if (a[3] == null)
```

It is also possible to explicitly assign null to a variable, for example

```
String s = null;
```

Here's an example of a method that might use the keyword `null`:

```
static boolean findString(String[] a, String s) {

    for (int i=0; i<a.length; ++i) {
        if (a[i] != null) {
            if (a[i].equals(s)) return true;
        }
    }
    return false;
}
```

The goal of this method is to search for a string `s` in array `a` and to return true if it is found. The code takes note of the possibility that some elements of the array might be missing, perhaps because only part of the array was filled in during a previous step, or perhaps because one or more elements were set to null. This method checks whether each string `a[i]` is null and compares it to `s` only if the string really exists. This is a critical check, because `a[i].equals(s)` will cause an exception if `a[i]` is null.

We'll talk much more about *null pointers* in the coming chapters.

It is possible to initialize an array as part of its declaration. Here are some examples:

```
int[] a = {4, 5, 6};
String[] b = {"This", "is", "fun", "isn't", "it?"};
int[][] m = {{1, 2, 3}, {4, 5, 6}};
```

The first two examples cause arrays to be allocated and initialized with the given values. The third example is a little trickier. Matrix `m` is effectively a two-dimensional array with two rows and three columns. Another way to declare `m` would be to write

```
int[] m = {{1, 2, 3}, a};
```

Array `a` would become, in every sense, the second row of `m`. This would mean, for example, that any change to `a[0]` would change `m[1][0]`. Although this last example illustrates the power of array initialization, it introduces a complexity that you'll probably want to avoid. It's always safe, and it's often appropriate, to initialize arrays using constants.

Chapter 10

Recursion

Some problems can be solved elegantly by using *recursion*. This is the idea of using methods that call themselves, either directly or indirectly.

Consider, for example, the factorial function:

$$f(n) \;=\; n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1.$$

Note that for $n > 1$,

$$f(n) \;=\; n \cdot f(n-1).$$

This version of the function is the basis of the following method for computing factorials:

```
public static int factorial (int n) {
    if (n <= 1) return 1;
    else return n * factorial(n-1);
}
```

Notice that the method calls itself with a different parameter. Just as any method is "put on hold" when it calls another, the execution of this method with parameter `n` > 1 is "put on hold" while it calls itself with parameter `n-1`.

Let's consider what happens when `n` is 10. The execution of `factorial(10)` pauses while a call is made to `factorial(9)`. The execution of `factorial(9)` pauses while a call is made to `factorial(8)`. And so on. Eventually a call is made to `factorial(1)`. At this point calls are pending with all choices of `n` from 10 down to 1. The call to `factorial(1)` returns 1 back to `factorial(2)`. The call to `factorial(2)` returns the result of `factorial(1)` by 2. This yields 2, which is passed back to the call to `factorial(3)`. The call to `factorial(3)` multiplies

the result by 3 and passes it back to `factorial(4)`. And so on. The result is that `factorial(10)` indeed yields

$$10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1.$$

10.1 The Fibonacci Sequence

The Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Each term is the sum of the two preceding it. Let's say that $fib_0 = 1$, $fib_1 = 1$, $fib_2 = 2$, $fib_3 = 3$, and so on. Here's how we can compute this function with a recursive method:

```
public static int fib (int n) {

    if (n <= 1) return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Again, if `n` is more than one, a call to `fib(n)` must wait for the completion of calls to `fib(n-1)` and `fib(n-2)`. Each *activation*, an individual call of the method, will have its own set of local variables.

This way of computing Fibonacci numbers is actually very inefficient. A call to `fib(10)` yields calls to `fib(9)` and `fib(8)`. The call to `fib(9)` yields another call to `fib(8)`, along with a call to `fib(7)`. The second call to `fib(8)` is wasted computation, and the waste is magnified enormously on later calls. An *iterative* (loop-based) solution is far more efficient:

```
public static int fib (int n) {

    int prev = 1;
    int curr = 1;

    for (int i=2; i<=n; ++i) {
        next = prev + curr;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

The lesson here is that recursion must be used carefully to avoid inefficiency. It's important to avoid unnecessary recalculation of intermediate results. The recursive

method for computing factorials did not suffer from this problem, and its efficiency is comparable to that of an iterative solution.

10.2 Towers of Hanoi

The Towers of Hanoi puzzle consists of n disks and a rack with three pins. The disks each have different diameters, with disk 1 being the smallest and disk n the biggest. Initially all of the disks are on the first pin, with the biggest disk at the bottom and the smallest at the top. The challenge is to move all of the disks to the second pin, while following two rules:

1. Only one disk can be moved at a time.

2. No disk can be placed on top of a smaller disk.

We would like to have a method that solves this puzzle and prints a list of moves. The method might use this header

```
public static void hanoi (int n, int from, int to)
```

The parameters specify the number of disks, the number of starting disk (1, 2, or 3), and the number of the ending disk.

Figure 10.1 shows recursive implementation of the `hanoi` method. The idea is that we can move a stack of n disks from one pin to other by first moving $n-1$ disks to the third pin (getting them out of the way), by moving the biggest one directly to the target pin, and then by moving the stack of $n-1$ disks to the target pin. Figure 10.2 shows the result when we call `hanoi (4, 1, 2)`.

Interestingly, this problem does not have a reasonable solution based on iteration, while recursion leads to a simple elegant solution.

10.3 Infinite Loops in Recursion

Recursion can lead to problems if used incorrectly. Consider the following:

```
public static int silly (int n) {    // a bad example
    return n * silly(n-1);
}
```

This is similar to the factorial method, but note that there is no "base case." A call to `silly(n)` will lead to an infinite sequence of calls and will essentially yield an infinite loop. Each call uses some memory, and eventually the loop will trigger a `StackOverflowError`, indicating that there is no more memory available for method calls. An *error* is similar to an exception.

```
public static void hanoi (int n, int from, int to) {

    if (n == 1)
        System.out.println ("Move disk 1 from pin " + from
                            + " to pin " + to);
    else {
        int other = 6 - from - to;
        hanoi (n-1, from, other);
        System.out.println ("Move disk " + n + " from pin "
                            + from + " to pin " + to);
        hanoi (n-1, other, to);
    }
}
```

Figure 10.1: A method for the Towers of Hanoi

```
Move disk 1 from pin 1 to pin 3
Move disk 2 from pin 1 to pin 2
Move disk 1 from pin 3 to pin 2
Move disk 3 from pin 1 to pin 3
Move disk 1 from pin 2 to pin 1
Move disk 2 from pin 2 to pin 3
Move disk 1 from pin 1 to pin 3
Move disk 4 from pin 1 to pin 2
Move disk 1 from pin 3 to pin 2
Move disk 2 from pin 3 to pin 1
Move disk 1 from pin 2 to pin 1
Move disk 3 from pin 3 to pin 2
Move disk 1 from pin 1 to pin 3
Move disk 2 from pin 1 to pin 2
Move disk 1 from pin 3 to pin 2
```

Figure 10.2: The result of calling `hanoi(4, 1, 2)`

10.4 More Examples

We close with several more examples of recursion. Each of these methods could also have been written using iteration instead of recursion. In coming chapters we'll see more examples where recursion could be used but iteration could not.

Here's an amusing way to multiply two integers:

```
public int mult (int a, int b) {
    if (a > 0) return mult(a-1, b) + b;
    else if (a == 0) return 0;
    else return mult(-a, -b);
}
```

Here's a way to raise an integer n to the k^{th} power (assuming k is non-negative):

```
public int pow (int n, int k) {   // we assume k >= 0
    if (k == 0) return 1;
    else return n * pow(n, k-1);
}
```

This approach takes k multiplications to do the exponentiation. Here's an alternative version that's more efficient:

```
public int pow (int n, int k) {   // we assume k >= 0
    if (k == 0)
        return 1;
    else if (k % 2 == 0)
        return pow(n*n, k/2);
    else
        return n * pow(n, k-1);
}
```

For large k, this takes many fewer multiplications, because it makes recursive calls using a sequence of k values that rapidly approaches 0.

Chapter 11

Exceptions

We've encountered several different kinds of exceptions so far, including:

- `ArithmeticException`: this occurs if you do integer division by zero
- `ArrayIndexOutOfBoundsException`: this occurs if you index beyond the bounds of an array
- `java.util.InputMismatchException`: this occurs if you do a `keyboard.nextInt()` and the user types something other than an int.

These are all variants of a `RuntimeException`, which is itself a variant of an `Exception`. We won't worry about the distinction between `RuntimeException` and `Exception` at this point.

11.1 Catching Exceptions

Here's an example of how you could "take control" of an exception:

```
int i;

try {
    i = keyboard.nextInt();
    System.out.println ("The nextInt() was successful");
}
catch (java.util.InputMismatchException e) {
    System.out.println ("It's an exception!");
}
```

This illustrates a *try-catch statement* in which a *try block* is associated with one or more *catch blocks*. We'll talk about how multiple catch blocks work in a bit. In the simple situation given here, we try to do the code in the try block. If an `InputMismatchException` ever occurs, control immediate jumps to the catch block, i.e. it won't reach the "successful" message. The variable `e` refers to an object that has further information about the exception; we won't worry about it here.

Here's an example of how you might really use exception handling:

```
private static int getMonth() {

    while (true) {
        try {
            System.out.print ("Enter a month [1-12]: ");
            int m = keyboard.nextInt();
            if (m >= 1 && m <= 12)
                return m;
            System.out.println ("Bad answer... try again.");
        }

        catch (java.util.InputMismatchException e) {
            System.out.println ("Bad answer... try again.");
            keyboard.nextLine();     // to clear the rest of the line
        }
    }
}
```

This code will gracefully handle any non-numeric inputs from the user. Note that the "true" condition on the while loop means that the loop will continue forever, or at least until the return is made within the try block. We'll talk about the use of `nextLine()` for line clearing in Chapter 12.

11.2 Using Multiple Catches

A try block can have multiple catches. Consider this example:

```
try {
    .
    .
    .
}
catch (java.util.InputMismatchException e) {
    .
    .
    .
}
catch (ArithmeticException e) {
    .
    .
    .
}
catch (RuntimeException e) {
    .
    .
    .
}
```

If an exception occurs within the try, the first applicable catch is used. Suppose it's an `InputMismatchException`. Recall that a `InputMismatchException` is a variant of a `RuntimeException`. Either catch could be used; the `InputMismatchException` one is first so it is used. If the three catches were reordered so that the `RuntimeException` case was listed first, neither of the other two would ever be run.

If an exception occurs and there is no corresponding catch block, the method terminates and the exception is passed up to whatever method called it. If the second method has a catch for that kind of exception, it is used, and otherwise the exception is passed further up the calling sequence. If unrolling the sequence of calls doesn't lead to a method with a catch, the program halts. If a program halts because of an exception, you should check to see if the message tells you where the exception occurred. A mention of something like `Lab5.java:43` probably means that the exception occurred on line 43 of `Lab5.java`.

11.3 Using Exceptions to Create More Robust Code

Remember the examples in Chapter 8 in which we were trying to convert a month number into a month name? One approach returned appropriate names when the number was 1 through 11 and returned "December" in the default case. That's actually not a very robust approach. Here's a better way:

```
private String monthName (int month) {

    switch (month) {
    case 1:
        return "January";
    case 2:
        return "February";
        .
        .
        .
    case 12:
        return "December";
    default:
        throw new RuntimeException ("Bad month number");
    }
}
```

Suppose the programmer somehow fails to ensure that this method is called only with a valid month number. It would be unfortunate if this method simply returned "December" without catching the error. The new default case uses a *throw statement* to generate a new exception. This will cause the program to crash (unless you're catching the exception), which is probably better than returning a bad result. We'll talk more about the details of a throw later.

The reason for having exceptions is to indicate that something abnormal has happened and to give program a chance to do something about it. Almost always, exceptions are thrown in some method (let's call it `f`) and caught in some other method that directly or indirectly calls `f`. This means that `f` doesn't have to give a return value in the way that it ordinarily would. Throwing the exception lets execution jump out of `f`, terminating the running of `f` and perhaps other methods.

11.4 Checked Exceptions

A *checked exception* is a kind of exception a programmer is required to deal with, or to at least notice. We will meet an exception of this kind, a `FileNotFoundException`,

in the next chapter. We will talk later about how you go about creating both checked and ordinary unchecked exceptions. For now, it suffices to understand that there is a difference and that special rules apply to checked exceptions.

If a checked exception is thrown in a method `f`, that method must either:

1. catch the exception, or

2. have a label that declares that that kind of exception might be thrown.

Here's an example. Let's suppose that `MyCheckedException` is a checked exception. The header for `f` would appear as follows, assuming that `f` itself didn't use a try-catch to catch that kind of exception:

```
static int f (int whatever) throws MyCheckedException {
```

The keyword `throws` here is a bit of a misnomer. It should really be interpreted as "might throw".

Now suppose that method `g` calls `f`, a method labelled with `throws MyCheckedException`. Method `g` must either catch the exception, or must itself be labelled with `throws MyCheckedException`.

It is possible, although not too common, for a main method to be labelled with a throws clause. In that case an uncaught exception would terminate the program and would produce error messages of the usual kind.

Chapter 12

Input and Output

In this chapter we explore more ideas about the ways in which programs can do input and output.

12.1 Scanner

Java's `Scanner` class—its full name is `java.util.Scanner`—offers a convenient way to read values from a keyboard, file, or other input source. Recall that by including the line

```
public static Scanner keyboard = new Scanner(System.in);
```

in the file for our main class, we are able to enable reading from the keyboard. Here is a little more detail about what happens.

The `Scanner` class usually works by breaking input into *tokens* that are separated by *white space*. White space is any sequence of one or more space, tab, or newline characters. Consider the following line:

```
3 56 a+b   "hi there" -3.2 67a
```

There are seven tokens in this line: three that look like numbers (`3`, `56`, and `-3.2`), one that looks like an expression (`a+b`), one that looks like the beginning of a string (`"hi`), one that looks like the end of a string (`there"`), and one containing digits and letters (`67a`).

The methods in the `Scanner` class permit programmers to read from the keyboard in the following ways:

- `keyboard.next()`: This method skips any white space and returns the next token as a `String`. After a call to this method, the character immediately

following the token will be available for the next attempt at reading. Beginning programmers rarely use this method directly, relying instead on one of the methods listed below.

- `keyboard.nextInt()`: This method calls `keyboard.next()` to obtain a token, and then it tries to convert the token to an int. The conversion will succeed if the token contains only digits (and perhaps a leading minus sign) and if the value is in the range -2^{31} to $2^{31} - 1$. If the conversion succeeds, the int is returned, and otherwise an `InputMismatchException` occurs. After a call to this method, the character immediately following the token will be available for the next attempt at reading.

- `keyboard.nextDouble()`: This method works in a similar way and returns a double. It will throw an exception if the next token in the input isn't an int or double.

- `keyboard.nextLine()`: This method reads characters, up to the next newline character, and returns them as a string. The newline character itself is not placed in the string. This method might have been called `nextString()`.

Now consider the following code:

```
System.out.print ("Enter three integers:");
int a = keyboard.nextInt();
int b = keyboard.nextInt();
int c = keyboard.nextInt();
keyboard.nextLine();            // this clears the newline character
System.out.print ("Now enter a string:");
String s = keyboard.nextLine();
System.out.println ("Here's your string: " + s);
```

Suppose the user types:

```
53 6
7
hi
```

The three integers will be placed in the three integer variables, and the word "hi" will be placed in `s`. Note the use of `keyboard.nextLine()` immediately after the third integer is read. This clears the newline character after the 7, along with anything else that appears on that line. If you are trying to read a mixture of numbers and strings, you'll usually need to do a `keyboard.nextLine()` to clear a line if: 1) you've just read a number, and 2) you're about to read a string.

12.2 Reading From Files

You can use `Scanner` objects to read from files. Here's an example:

```
Scanner infile = new Scanner(new java.io.FileReader("myfile"));

int i=infile.nextInt();
int j=infile.nextInt();

System.out.println ("Values " + i + " and " + j + " were read.");
```

All of the various reading methods discussed above can be used to read from an arbitrary `Scanner`.

When you try to create a `FileReader` in the code above, a `java.io.FileNotFoundException` will occur if the file doesn't exist or can't be read. Here's a method you might write to open a file and catch the exception:

```
private static Scanner openFile(String s) {
    try {
        Scanner infile = new Scanner(new java.io.FileReader(s));
        return infile;
    }
    catch (java.io.FileNotFoundException e) {
        System.out.println ("Error opening file for reading");
        System.exit(1);
        return null;        // this stmt will never be reached
}
}
```

`FileNotFoundException` is a checked exception, so you are obligated to either catch the exception or mark the method with `throws FileNotFoundException`.

There are special rules concerning what happens when the end of a file is reached. If you try to read a line, a `java.util.NoSuchElementException` occurs. To avoid the possibility of causing an exception, you can call the `Scanner` method `hasNextLine()`, which returns a boolean. It returns true if a call to `nextLine()` will succeed.

Here's how you can count the lines in a file:

```
private static int countLines(String filename) {

    Scanner infile = openFile(filename);
    int count = 0;

    while (infile.hasNextLine()) {
        String s = infile.nextLine();
        count++;
    }

    infile.close();       // make the file available for reuse
    return count;
}
```

Note the use of the method `close()` once you are done using the `Scanner`. This is a way to tell the system that you're done reading the file. Doing a `close()` helps free up system resources and can help avoid errors resulting from having too many files in use at once.

If you do a `nextInt()` or a `nextDouble()` at the end of a file, a `java.util.NoSuchElementException` is also thrown. You can use methods `hasNextInt()` and `hasNextDouble()` to determine if the `nextInt()` and `nextDouble()` methods will succeed.

Here's a method to add up all the integers in a file:

```
private static int addNumbers (Scanner infile) {

    int sum = 0;
    while (infile.hasNextInt()) {
        sum += infile.nextInt();
    }
    infile.close();
    return sum;
}
```

12.3 Output to Files

You can use the `print()`, `println()`, and `printf()` methods to write to a file by using a `PrintWriter` object. Here's an example:

```
import java.io.*;
import java.util.Scanner;

public class Copy {

    public static void main (String[] args) throws FileNotFoundException {

        Scanner infile = new Scanner(new FileReader("studentlist"));
        PrintWriter outfile = new PrintWriter("myoutfile");

        while (infile.hasNextLine()) {
            String s = infile.nextLine();
            outfile.println(s);
        }
        infile.close();
        outfile.close();
    }
}
```

There are several important features here:

1. By prefacing the file with the line

   ```
   import java.io.*;
   ```

   we are able to use the names `FileReader`, `PrintWriter`, and `FileNotFoundException` without giving the prefix `java.io.`

2. We are using `throws FileNotFoundException` as part of the declaration of `main()` as an alternative to catching this kind of exception.

3. **It is vital to do `close()` on the `PrintWriter` when you are done.** Failure to do this might mean that the file won't contain everything that it was supposed to include.

4. The data that you write to a file is sometimes buffered in the computer's memory and is not necessarily written to the file right away. Disk writes are

very slow compared to processor speeds, and many thousands of instructions can be executed in the time needed to do one write. Collecting data until there is a large amount that can be written all at once can lead to huge speedups in running times. Sometimes, however, you'd like data to be written to disk immediately, for example if you want to use another program to inspect the file while your program is still running. In this case, you can use the `flush()` method on the `PrintWriter` to cause any buffered data to be written to the disk. If you do this after each `print`, `println`, or `printf`, you'll ensure that the file is kept up to date.

12.4 Setting the Delimiter in a Scanner

A `Scanner` usually uses white space (one or more spaces, tabs, and newline characters) as the delimiter that is used to separate tokens. Sometimes it is useful to specify a different choice of delimiter. For example, suppose you are trying to extract words out of a document. You might know that these items should be included as part of the delimiter:

1. Any white space character.
2. These punctuation characters: `. , ; :`
3. The pattern `---`

You can specify the delimiter by writing:

```
Scanner infile = new Scanner(new FileReader("myfile"));
infile.useDelimiter("(\\s|[.,;:]|---)+");
```

After the delimiter pattern is set, calls to `infile.next()` will yield the desired sequence of words.

The encoding of the delimiter pattern is obviously a bit opaque. The string `\\s` means "any white space character", and the string `[.,;:]` means any of the four punctuation characters. The `|` characters within the parentheses mean "or", so the whole parenthesized expression means "white space or a punctuation character or `---`". The plus sign at the end means that a full delimiter must consist of one or more substrings matching the pattern given in the parentheses.

Full information on possible pattern strings can be obtained by looking at the API documentation for class `java.util.regex.Pattern`. There's a lot there!

12.5 Creating Scanners from Strings

Suppose you are writing a program in which the user is asked questions to which he or she should reply with a single lines of input. Suppose further that you'd like to use `nextInt()` or `nextDouble()` calls within each line but you'd like to avoid the possibility that those calls will affect the processing of the next line. This is a little trickier than it seems. You might try using this code:

```
while (true) {
    System.out.println("Enter a line with three ints:");
    for (int i=0; i<3; ++i) {
        doSomething(keyboard.nextInt());
}
```

Now consider what happens if the user types only two ints and then hits return. The program will get stuck trying to read the third int and won't prompt for the second line of input. If the user does decide to type more data, the first token will be used as if it had been typed on the first line.

There are lots of times when you are writing a program for your own use and are willing to put up with the possibility of strange behavior if enough data is not typed. But let's consider alternatives.

One idea that doesn't work is to try to add a call to `hasNextInt()`. For example:

```
while (true) {
    System.out.println("Enter a line with three ints:");

    int i;
    for (i=0; i<3 && keyboard.hasNextInt(); ++i) {
        doSomething(keyboard.nextInt());
    if (i<3)
        System.out.println ("I asked for three!");
}
```

The hope here is that `hasNextInt()` will eventually be false if too little data was given. It turns out that `hasNextInt()` will itself get stuck if the user hasn't given three ints. It will wait to see if the user ever types the desired int, even on a later line.

An approach that does work is the following:

```
while (true) {
    System.out.println("Enter a line with three ints:");
    String theLine = keyboard.nextLine();
    Scanner s = new Scanner(theLine);
    int i;
    for (i=0; i<3 && s.hasNextInt(); ++i)
        doSomething(keyboard.nextInt());
    if (i<3)
        System.out.println ("I asked for three!");
}
```

In this code, `Scanner s` reads material from a string rather than from the keyboard.

The idea of using a `Scanner` to read from a string can of course be applied when the string comes from a file or is created in any other way.

12.6 Reading a Single Character from a Scanner

While the `Scanner` class has a `nextInt()` and a `nextDouble()` method, there is no method to read a single character from the `Scanner`. One way to do this is by writing

```
String s = infile.findInLine(".");
```

This will read a single character from the current line in `Scanner infile`, regardless of the delimiter pattern, and will yield a string containing that character. If the end of the current line has been reached, the string will equal null. In this case it might be appropriate to call `hasNextLine()` to check if there is another line and then `nextLine()` to move to it. Since you're already at the end of the line, the call to `nextLine()` will simply clear the newline character and get you ready to read the next line.

The pattern given in the call to `findInLine` matches any single character except a newline character.

Chapter 13

Arrays and References

In this chapter we consider precisely how arrays are handled in Java.

Suppose you declare

```
int[] a = new int[100];
```

This statement does several things:

1. It creates a variable of type `int[]`, which means "array of int". The variable won't contain the actual array, but rather it will contain a *reference* (also called a *pointer*) to the array itself. Think of a reference as a numeric address telling where the array is located in the computer's memory.

2. It allocates space to hold 100 integers.

3. It sets the variable so that it refers to the allocated memory.

Now suppose you execute a statement such as

```
a[3] = 27;
```

The computer will check variable `a` to obtain the address of the array. It will then go to the array itself, find position 3, and store the value 27 in that position.

Now suppose you use the following statements:

```
int[] b;
b = a;
System.out.println(b[3]);
```

These statements cause the number 27 to be printed! This happens because the address found in variable `a` is copied to variable `b`. That causes `a` and `b` to point to the very same array. Printing `b[3]` therefore prints the value that we just stored in `a[3]`.

13.1 Comparing References

The statement

```
if (a == b)
```

compares the values in variables `a` and `b`. Because these variables contain references, the effect is simply to ask "Do `a` and `b` refer to the same array?" Notice that this is different than asking if `a` and `b` refer to arrays containing the same set of values. This second question could be solved by this method:

```
private static boolean compareArrays (int[] a, int[] b) {

    if (a.length != b.length) return false;
    for (int i=0; i<a.length; ++i)
        if (a[i] != b[i]) return false;
    return true;
}
```

13.2 Passing Array References as Parameters

Recall that when you call a method, the parameter variables are initialized with copies of the value given when the method is called. This "call-by-value" system means that changes made within the method to parameter variables don't affect the variables of the calling method. Here an example of a method that doesn't work the way you might suppose:

```
private static void swap (int i, int j) {   // a useless method
    int t = i;
    i = j;
    j = t;
}
```

This method swaps the contents of variables `i` and `j`, but unfortunately it's useless. Suppose you write

```
swap (m, n);
```

The swap method runs with variable `m` copied into `i` and variable `n` copied into `j`. Because `i` and `j` are copies, `m` and `n` are unaffected when they are swapped.

Now consider this code:

```
public static void swapElements (int[] a, int i, int j) {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

If you call

```
swapElements(myArray, x, y);
```

the elements in positions `x` and `y` of `myArray` will be swapped. Let's see what happens. Parameter `a` receives a copy of the variable `myArray`. Variable `myArray` is a reference to an array, so `a` ends up referring to the same array. (Recall that this means that `a` gets the address of the array itself.) By swapping elements in the array referenced by `a`, the method effectively swaps elements in the array referenced by `myArray`.

13.3 More Examples with Arrays

The following method reverses the contents of an array:

```
private static void reverse (int[] a) {

    // This methods reverses the elements of array a,
    // so the first element becomes the last, and so forth.

    int i=0;
    int j=a.length - 1;

    while (i < j) {
        swapElements (a, i, j);
        i++;
        j--;
    }
}
```

If you call `reverse(b)`, the elements of `b` will be reversed.

This method returns a new array that is a copy of the original one.

```
private static char[] copy (char[] a) {

    // This method creates a copy of array a

    char[] b = new char[a.length];

    for (int i=0; i<a.length; ++i)
        b[i] = a[i];
    return b;
}
```

You can write

```
char[] c = copy(a);
```

assuming that `a` is an existing array of characters.

This method sums the values in an array:

```
private static double addPos (double[] d) {

    // This method adds up the values in an array.

    double sum = 0.0;

    for (int i=0; i<d.length; ++i)
        sum += d[i];

    return sum;
}
```

This method zeroes the elements in a particular range of positions in an array:

```
private static void zeroRange (double[] d, int lo, int hi) {

    // This method sets elements lo..hi of d to zero.

    for (int i=lo; i<=hi; ++i)
        d[i] = 0;
}
```

This method looks through an array, examining values in positions `lo` through `length-1`, and returns the position of the smallest element:

```
private static int findMinPos (int[] a, int lo) {

    // This method looks in the elements of a in positions
    // greater than or equal to "lo" and returns the index
    // of some element of minimum value.

    int pos = lo;

    for (int i=lo+1; i<a.length; ++i)
        if (a[i] < a[pos]) pos = i;

    return pos;
}
```

This method generates a list of the first `n` primes:

```
private static int[] primelist (int n) {

    // This method creates a list of the first n prime numbers.
    // It assumes the existence of a method primetest()
    // that checks primality.

    int[] list = new int[n];

    int i=2;       // the next number to check
    int count=0;   // the number of primes found so far

    while (count < n) {

        if (primetest(i)) {
            list[count] = i;
            count++;
        }
        i++;
    }
    return list;

}
```

13.4 Sorting

Sorting is the problem of taking an array and rearranging the elements so that they follow some order. Here's a simple approach to sorting, called "selection sort." In this example, we'll assume that we're working with simple integers and that we want to sort them in increasing order:

```
private static void selectionSort (int[] a) {

    for (int i=0; i<a.length; ++i) {
        int minPos = findMinPos(a, i);
        swap (a,i,minPos);
    }
}
```

Note that this method uses two of the methods we saw earlier. The idea is to work our way through the array, determining which element should go in each position, and then swapping appropriately.

This approach is simple and fairly efficient, at least for relatively small arrays. There are other schemes, such as *mergesort* or *quicksort*, that are more complex but much more efficient if your array contains more than a few thousand elements. (If you've ever heard the words *bubble sort*, please forget them now. They refer to an attractive but inferior way to sort data that is often taught to beginning programmers.)

13.5 Array Allocation and Deallocation

When you create a new array with the `new` keyword, we say the array is created is *dynamically allocated*. The array doesn't disappear when the method ends. Instead it persists and can be returned via a return value, passed to other methods and so on. The array will remain in memory as long as there is some reference to it, i.e. as long as there is some variable holding its address. When there are no more references the array is *garbage-collected* and its memory becomes available for reuse.

Here's an example showing a case where an array becomes available for garbage collection:

```
int[] a = new int[10];
int[] b = new int[20];
a = b;
```

After the first two statements, variables `a` and `b` point to distinct arrays. The third statement causes the address in `b` to be copied into `a`, meaning that both variables

point to the length-20 array. No variable points to the shorter array, so the shorter array is now garbage. When garbage collection occurs, the space used for the shorter array becomes available for reuse.

13.6 Null References

Recall the keyword `null`, which was introduced in section 9.4. An array reference variable can be assigned the value null in a variety of ways. Consider, for example, the following:

```
int[] a = null;
int[] b = a;
int[][] c = new int[10][];
```

This code sets both `a` and `b` to be null. Furthermore, each element of the one-dimensional array `c` is also null. (Each variable `c[i]` has type `int[]` and value null.)

Null represents a reference to a non-existent array. It's fine to copy a null reference, for example when we write `b=a`, or to compare it, for example when we write

```
if (a == null)
```

On the other hand, it is an error to try to use a null reference to access the array to which it supposedly points. Here are some examples, each of which causes a `NullPointerException`:

```
System.out.println (a[3]);
System.out.println (b.length);
System.out.println (c[1][2]);
```

The words "pointer" and "reference" are synonyms when they are used to describe a kind of value or variable. While other languages routinely use the word "pointer," Java uses the word "reference" almost exclusively. The notable exception occurs in the name `NullPointerException`.

Chapter 14

Enumerated Types

Suppose we are trying to keep track of a deck of cards, where we are representing each suit by an int from 0 to 3 and each value by an int from 2 to 14. It doesn't really matter how the ints are associated with the four suits. Let's suppose that the value 11 means Jack, 12 means Queen, 13 means King, and 14 means Ace.

Here is how we could initialize an unsorted deck:

```
int[] suits = new int[52];
int[] values = new int[52];

int k=0;
for (int s=0; s<4; ++s) {
    for (int v=2; v<=14; ++v) {
        suits[k] = s;
        values[k++] = v;
    }
}
```

We could get a string describing each card with the following code:

```
static String[] suitNames = {"Spades", "Hearts", "Clubs", "Diamonds"};
static String[] faceCards = {"Jack", "Queen", "King", "Ace"};

static String getCardName (int s, int v) {
    if (v <= 10)
        return v + " of " + suitNames[s];
    else
        return faceCards[v-11] + " of " + suitNames[s];
}
```

This isn't bad, but it offers some complications. First, the programmer will have to keep in mind at all times the fact that suit 0 means Spades, and so on. Furthermore, suppose he or she writes the following:

```
System.out.println (getCardName(values[i],suits[i]));
```

Note that the value is being passed where the suit is expected, and vice versa. This code will compile correctly, but incorrect behavior will occur at runtime.

An *enumerated type* is a way to define a new type that can represent one of a finite set of alternatives. We'll use an enumerated type to represent a suit.

Suppose a file `Suit.java` contains the following:

```
public enum Suit {Spades, Hearts, Diamonds, Clubs};
```

This code says that a variable of type `Suit` can take on one of four values. We could then write the following code in another class.

```
Suit s = Suit.Spades;
Suit t = Suit.Hearts;

System.out.println ("The suits are " + s + " and " + t);
if (s.equals(t))
    System.out.println ("The suits match.");
```

The output would be

```
The suits are Spades and Hearts
```

In doing the assignment into variables `s` and `t`, we had to say `Suit.Spades` and `Suit.Hearts` because `Spades` and `Hearts` were symbols defined only within the enumerated type `Suit`.

The code for creating a deck of cards could then become the following:

```
Suit[] suits = new Suit[52];
int[] values = new int[52];

int k=0;
for (Suit s : Suit.values()) {
    for (int v=2; v<=14; ++v) {
        suits[k] = s;
        values[k++] = v;
    }
 }
```

14.1 For-Each Loops

The outer `for` loop in the code above is a *for-each* loop. The loop will run four times, with variable `s` taking on each of the different values in the enumerated type.

For-each loops can be used with arrays, too. Here's an example:

```
String[] a = ...;

for (String t : a)
    ...
```

The body of the loop will run once for each element of `a`, starting in position 0, with `t` taking on the value of each element.

For-each loops are slightly more succinct way to iterate across the elements of an array and are appropriate when 1) you want to iterate across the whole arrays, and 2) you don't need access to the index of the current element.

14.2 Switch Statements on Enumerated Types

Switch statements can be used with enumerated types. Here's an example:

```
Suit s = ...;

switch (s) {
case Spades:
case Clubs:
    System.out.println ("It's a black card.");
    break;
case Diamonds:
    System.out.println ("It's a diamond.");
    break;
default:
    System.out.println ("It's something else, I'm guessing hearts");
}
```

Of course, it's more than a guess that the suit is hearts in the default case, because a value of type `Suit` must have one of the designated values. Note that it was possible to omit the `Suit.` prefix for the suit names in this example. The use of a `Suit` variable in the switch implied that the cases would be suits and made the prefix unnecessary.

14.3 Compiling and Running Programs with Multiple Files

To compile a program involving more than one `.java` file, the best approach is to create a new directory to hold them. If you are using Unix, you can then go to that directory and issue the command

```
javac *.java
```

which compiles all of the `.java` files in the directory. To execute the program, you can then run

```
java MainClass
```

assuming that `MainClass` is the class containing the method `main`.

Chapter 15

Writing Classes to Define Objects

Programmers can use classes to define their own objects. You've actually already done this, because an enumerated type is a simple class that defines a simple kind of value.

Let's begin with an example of how a program might use a more complex kind of object, and then we'll look at how we'd write the class for the object. The program in Figure 15.1 uses a `BankAccount` object. This program probably isn't too hard for you to read. We create two bank accounts, one for Joe and one for Jane, each with some starting balance. We make a deposit to one account with the `makeDeposit()` method, and then we go on to deduct funds, get a balance, and print information about an account. Because the main method is in a class called `BankTest`, it must appear in the file `BankTest.java`.

Figure 15.2 shows how we could define a bank account and the methods that can be used with it. Because the class is named `BankAccount`, its code would appear in a file called `BankAccount.java`.

15.1 Instance Variables

In `BankAccount`, there are two *instance variables* defined: `owner` and `balance`. These are variables that are outside of any particular method and that are associated with the object itself. Every bank account has an owner and a balance. While the value of these variables can change, the fact that an account has some owner and some balance will not change. Put differently, the values in the instance variables characterize the object.

Instance variables are never labelled as `static`. In this case, the instance vari-

```
public class BankTest {

    public static void main (String[] args) {

        BankAccount his = new BankAccount ("Joe Smith", 23.00);
        BankAccount hers = new BankAccount ("Jane Doe", 94.00);

        his.makeDeposit (12.00);
        hers.deductFunds (20.00);

        System.out.println (his);
        System.out.println (hers);

        double balance = hers.getBalance();
        System.out.printf ("Jane's balance is $%.2f\n", balance);

        if (hers.deductFunds(80.00))
            System.out.println ("Funds are available");
        else
            System.out.println ("Funds are not available");
        System.out.println (hers);
    }
}
```

Figure 15.1: A class that uses a `BankAccount` object

```
import java.io.PrintWriter;
import java.io.StringWriter;

public class BankAccount {
    /////////////// instance variables //////////////////////////////////////
    private String owner;
    private double balance;

    /////////////// constructor ////////////////////////////////////////////
    public BankAccount (String o, double b) {
        owner = o;
        balance = b;
    }

    /////////////// public methods /////////////////////////////////////////
    public void makeDeposit (double amount) {
        balance += amount;
    }
    public boolean deductFunds (double amount) {
        if (checkAvailability(amount)) {
            balance -= amount;
            return true;
        }
        else return false;
    }
    public double getBalance() {
        return balance;
    }
    public String getOwner() {
        return owner;
    }
    public String toString () {
        StringWriter s = new StringWriter();
        PrintWriter p = new PrintWriter (s);
        p.printf ("Owner: %s, Balance = $%.2f", owner, balance);
        p.close();
        return s.toString();
    }

    /////////////// private method ////////////////////////////////////////
    private boolean checkAvailability (double amount) {
        if (balance >= amount)
            return true;
        else
            return false;
    }
}
```

Figure 15.2: Class `BankAccount`

ables are labelled `private`, meaning that they can not be accessed from outside class `BankAccount`. We'll talk more about this later.

15.2 Constructors

When an object is created by using the `new` operator, memory is allocated for the object. Part of this memory is used to hold the object's instance variables. After the memory is allocated, a *constructor* is run for the object. (Some classes have multiple constructors; we won't worry about that for now.) A constructor is simply a special method that is run when the object is created.

Look at the constructor in the code above. The header has several interesting features:

1. The name of the constructor matches the name of the class.
2. There is no keyword `static`.
3. There is no return type.

Constructors are often very simple. In this case, the constructor simply copies values from its parameters into the instance variables.

Now recall the lines from the main method that we used to create the two bank accounts:

```
BankAccount his = new BankAccount ("Joe Smith", 23.00);
BankAccount hers = new BankAccount ("Jane Doe", 94.00);
```

The values given within the parentheses on each line are assigned to the parameters of the constructor, which in turn copies them into the appropriate instance variables. In this way we create two correctly defined `BankAccount` objects. The variables `his` and `hers` are references to the objects, in the same way that array variables were really references to the actual arrays.

15.3 Public Instance Methods

The `BankAccount` class contains several methods that are public and that are not static. The keyword `public` is the opposite of `private`; it means that these can be called from methods in other classes. As discussed in section 5.2, the absence of the keyword `static` means that they are methods that must be applied to objects. When we wrote

```
his.makeDeposit (12.00);
```

in the main method, we requested that the method `makeDeposit` be applied to the object referenced by variable `his`. Because `his` is a `BankAccount`, the system will look for a method called `makeDeposit` in class `BankAccount`. It will execute the method using the instance variables of the particular object referred to by `his`. The method takes a value `amount`, given as a parameter, and uses it to update the instance variable `balance`. The result is that Joe's balance is increased by twelve dollars.

The method call

```
hers.deductFunds (20.00);
```

works similarly. Note, however, that `deductFunds` is a method returning a boolean. (The point of the boolean is to indicate whether or not the funds are available.) In this case, the call to `deductFunds` doesn't use the return value in any way. The second call to `deductFunds`,

```
if (hers.deductFunds(80.00))
```

does use the value in an if-then-else statement.

The final statement

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

prints out information about the object referenced by `hers`. Whenever you execute a statement like this, or whenever you concatenate an object to a string, the system looks for a public no-arguments instance method called `toString` in the object's class. If the method exists, it is executed to obtain a `string` describing the object.

The `toString()` method in `BankAccount` uses a new class, `StringWriter`, to help build a string. By creating a `PrintWriter` based on a `StringWriter`, you are able to use `printf` and other operations to write into a string. After doing `close()` on the `PrintWriter`, you can apply the `toString()` method to the underlying `StringWriter` to get the string that was built.

15.4 Private Instance Methods

Class `BankAccount` also contains a method called `checkAvailability` that is labelled `private`. A private method can only be called from elsewhere within the same class. In this example, `deductFunds` calls `checkAvailability` to test if a sufficient balance exists.

Methods should be labelled `private` if there is no good reason to permit methods in other classes to use them. By making them `private`, you can protect the object from incorrect manipulation by methods outside the class. We'll see more examples of this later.

15.5 Public Instance Variables

Although it doesn't happen in the bank account class, instance variables can also be labelled `public`. If an instance variable is public, it can be manipulated in other classes. For example, if variable `balance` were public, you could use or modify `his.balance` within class `BankTest`. By making it private, we reduce the possibility that the balance can be modified by mistake. The main method can still use `his.getBalance()`, a public method, to retrieve the balance, but it can't use the variable directly.

15.6 An Example Class: `Student`

Figure 15.3 shows an example of another class, `Student`. This class is similar to the `BankAccount` class in the previous section, in the sense that the main purpose of the class is to hold the data associated with some entity, in this case a student. There are private fields for the data itself, and public methods for accessing it. Note the presence of a `toString` method that produces a string describing the object. The `precedes` determines whether or not one student has a name that precedes another in alphabetic order. Here's how it might be used:

```
Student s = new Student ("Bill", 2002, 438);
Student t = new Student ("Jane", 2004, 610);
if (t.precedes(s)) ...
```

After creating the two `Student` objects, this code runs the `precedes` method on t. This means that when we refer to `name` within the method, we mean the name associated with `t`. The method compares `t`'s name with that of `s` by applying the `compareTo` method from class `String`. The latter method yields a negative value if `t`'s name comes first alphabetically, zero if the names are the same, and a positive value if `t`'s name comes later alphabetically. Putting this together, we see that the `precedes` method returns true if `t`'s name comes before that of `s`. Of course there's nothing special about the variables `t` and `s` here. If we had declared

```
Student u = ...
```

we could write

```
if (u.precedes(t)) ...
```

to test if `u`'s name precedes `t`'s name.

```
public class Student {

    private String name;
    private int year;
    private int box;

    public Student (String name, int year, int box) {
        this.name = name;
        this.year = year;
        this.box = box;
    }

    public String getName () {
        return name;
    }

    public int getYear() {
        return year;
    }

    public int getBox() {
        return box;
    }

    public boolean precedes (Student s) {
        if (name.compareTo(s.name) < 0) return true;
        else return false;
    }

    public String toString() {
        return name + " " + year + " " + box;
    }
}
```

Figure 15.3: Class `Student`

15.7 Keyword `this`

Within an instance method, you can use the keyword `this` in order to access the "current object." For example, if you execute the following statement,

```
if (u.precedes(t)) ...
```

then you can use `this` within `precedes` to access the object to which `u` points. Similarly, you can use `this` within a constructor to reference the object that is being constructed.

One effect of this definition is that you can always write `this.v` in order to refer to instance variable `v` in the current object. It's generally fine to do it either way.

The constructor of `Student` shows a case in which it's vital to use `this`. There is an instance variable called `year`, and there's a parameter with the same name. Java permits this situation, which is arguably dangerous. In any event, it's possible to copy the parameter into the instance variable by writing

```
this.year = year;
```

The right-hand side refers to the parameter, and the left-hand side refers to the instance variable. Programming languages typically give priority to more local definitions, which is what causes `year` to be associated with the parameter. This is one of the rare situations in Java in which two definitions of a name are permitted to coexist. It's possible to avoid the ambiguity by simply avoiding giving the same name to parameters and instance variables.

Within an instance method, you can write something like

```
this.f();
```

to call an instance method `f()` on the same object. The effect is identical to making the call without the `this.` prefix.

15.8 An Example Class: `StudentList`

Figure 15.4 shows an example of another class, `StudentList`.

Let's look carefully at what's happening in this code. There are three private instance variables, one that holds a reference to an array of `Student` references, one that holds an int giving the capacity of the array, and one telling how many students are actually represented in the array. The constructor creates the array, sets the capacity appropriately, and sets the initial count to zero.

Method `loadFromFile` takes a file name as a parameter. It opens the file, reads data from the file, and calls `addStudent` to add an appropriate `Student` object

```
public class StudentList {

    private Student[] list;
    private int capacity;
    private int count;

    public StudentList ( int n ) {
        list = new Student[n];
        capacity = n;
        count = 0;
    }

    public void loadFromFile (String filename) throws FileNotFoundException {
        Scanner infile = new Scanner(new FileReader(filename));
        while (infile.hasNextLine()) {
            String name = infile.nextLine();  // each name fills one line

            int box = infile.nextInt();   // other data is on the next line
            int year = infile.nextInt();
            infile.nextLine();   // clear the newline character

            addStudent (name, box, year);
        }
        infile.close();
    }

    public void print() { ////////////////////////////////////////////////
        for (int i=0; i<count; ++i)
            System.out.println (list[i]);
    }

    public Student findByBox(int box) { ///////////////////////////////////

        for (int i=0; i<count; ++i)
            if (box == list[i].getBox()) return list[i];
        return null;
    }

    public Student addStudent (String name, int box, int year) { /////////

        // This method assumes that the list is ordered alphabetically
        // according to the names of the students.

        Student s = new Student (name, year, box);
        if (count >= capacity)
            throw new RuntimeException ("Too many students");

        count++;

        int i;
        for (i=count-2; i>=0 && s.precedes(list[i]); --i)
            list[i+1] = list[i];
        list[i+1] = s;
        return s;
    }
}
```

Figure 15.4: Class `StudentList`

to the list. We'll talk about the `addStudent` method in a bit. For now let's just assume that it correctly creates `Student` objects, adds them to the list, and adjusts the count.

Method `print` prints information about the students in the list. Notice how simple it is. It uses a for loop to go through the list. It then does a `println` on each student. Recall that printing of objects works by printing the String obtained by invoking the `toString()` method on the object.

Method `findByBox` goes through the list checking for a student with the given box number. If the student is found, the method returns a copy of the reference for the student. If the student isn't found, it returns null. Here's a example of how you might use this method, assuming that `myList` is some `StudentList`:

```
Student s = myList.findByBox(100);
if (s == null)
    System.out.println ("No one has that box."):
else
    System.out.println ("That box belongs to " + s.getName());
```

Method `addStudent` adds a student to the array. It assumes that the existing list is ordered alphabetically according to student names, and it ensures that the new student is added in the correct position to maintain the order. The method begins by constructing the `Student` object and checking to make sure that there is room to add it to the list. If there is, it uses a loop to locate the right position for the new `Student`, shifting the elements of the array to make room for the new one.

15.9 Conditional Boolean Operators

Method `addStudent` also illustrates a feature of the `&&` operator, which is a "conditional and." Here is the for statement used above:

```
for (i=count-2; i>=0 && s.precedes(list[i]); --i)
```

As we discussed earlier, the `&&` operator will yield true if the operands on both sides of the operator are true. Suppose that the left operand is false. The overall result will surely be false, and the computer does not bother to evaluate the expression on the right. This behavior is often desirable. In the example above, the precedes test occurs only if `i >= 0`. That's good news, because if `i` is less than zero, the use of `list[i]` will cause an exception. The conditional behavior of the `&&` operator often helps avoid exceptions and meaningless tests. Obviously we need to be careful in ordering the operands of an `&&` operator. This statement is probably undesirable:

```
if (s.precedes(list[i]) && i>=0) ...      // bad example
```

Note that an exception will occur if `i` is ever negative. Here's another example of the correct use of `&&`:

```
if (s != null && s.precedes(t)) ...
```

The "conditional and" prevents us from applying the `precedes` method to a `null` pointer, which is erroneous and would lead to an exception.

The `||` operator works similarly and is a "conditional or". Here are two examples:

```
if (i<0 || list[i]==null) ...
if (s == null || s.precedes(t)) ...
```

The left operand is evaluated first; if it is false, the right operand is evaluated. While it is common to use and rely on the conditional aspects of the `&&` operator, it is less common to use the `||` operator in this way.

15.10 An Example Class for a Grid of Colors

Figure 15.5 shows a final example class.

Class `Grid`, which is associated with a graphical program, maintains a matrix of `Color` objects. (The class `Color` is defined in a Java package called `java.awt`; the import line lets us access the class.) Method `recolor` causes new random colors to be assigned to the elements of the matrix. Variable `count` is used to count the number of times the matrix has been colored. Note how the constructor is used to ensure that a `Grid` object is correctly initialized.

15.11 Summary

You should write a class to define a new kind of object when you have an entity or idea that has data and operations associated with it. Some objects do little more than hold data (for example, `BankAccount` and `Student` objects), while others (such as `StudentList` and `Grid` objects) are more complex. Constructors must ensure that the object is initially given some sensible state. For simple objects this sometimes means filling the instance variables with values copied from parameters, while for others there may be significantly more work.

```
import java.awt.Color;

public class Grid {

    public int size;
    public Color[][] color;
    public int count;

    public Grid (int size) {

        this.size = size;
        count = 0;
        color = new Color[size][size];
        recolor();
    }

    public void recolor() {
        count++;
        for (int i=0; i<size; ++i)
            for (int j=0; j<size; ++j)
                color[i][j] = randomColor();
    }

    private Color randomColor() {

        int red   = (int) (Math.random() * 256);
        int green = (int) (Math.random() * 256);
        int blue  = (int) (Math.random() * 256);

        return new Color (red, green, blue);
    }
}
```

Figure 15.5: Class `Grid`

Chapter 16

More on Classes

16.1 Class Variables and Methods

A class can contain variables that have declarations using the keyword `static`. These *class variables* are similar to instance variables, but there is only one copy of the variable for the whole class.

The most common use of class variables is in classes that contain only static methods, for example in a class containing a main method. Suppose you were building a one-class program with a variable `size` that is needed in all of the methods. You could do the declaration in the following way,

```
private static int size;
```

placing the declaration inside the class but outside of any method. The variable could then be accessed in any method in the class, without the need to pass it as a parameter.

It is regarded as bad style to use class variables in this way, because the flow of information between methods is obscured. If a class variable is inadvertently changed in one method, other methods will be affected. This is, of course, true when variables are passed as parameters or returned from methods, but the explicit use of parameters and return values makes dependencies more obvious.

Sometimes it is useful to use class variables in other ways. For example, recall the class `Student` that appeared in Figure 15.3. Suppose we wanted to keep track of how many `Student` objects had ever been created. We could declare a counter variable in the following way:

```
private static int studentCount = 0;
```

The constructor could then be the following:

```
public Student (String name, int year, int box) {
    this.name = name;
    this.year = year;
    this.box = box;
    studentCount++;
}
```

Each time that a `Student` object is created, the counter is incremented.

To permit other classes to retrieve the counter, which is in a private variable, we could add a *class method* to `Student`:

```
public static int getCount() {
    return studentCount;
}
```

A class method uses the keyword `static` and is associated with the class as a whole rather than with a particular object. Method `getCount` could then be called from another class by writing

```
Student.getCount()
```

Writing class methods was our topic way back in Chapter 6. Things are a little more complicated now that we have the possibility that some classes might have both static and nonstatic variables and methods.

Class methods are never permitted to call instance methods or to access instance variables, because class methods are not associated with an underlying object. Suppose, for example, that we referred to instance variable `name` in method `getCount()`. How would we know which student's name should be used? On the other hand, instance methods and constructors can freely call class methods or access class variables. We saw an example of this above, when the constructor for `Student` accessed variable `studentCount`.

16.2 Initializers

All instance variables and class variables are assigned initial values. The default initial values are those described for array elements in section 9.4, either 0, 0.0, false, `'\0'`, or null, depending on the type of the variable.

The presence of default initial values distinguishes class variables and instance variables from local variables. Recall that any attempt to use an uninitialized local

variable causes an compilation error. The same situation does not hold for class variables and instance variables.

Individual initialization values can be assigned to class and instance variables. Here are some examples:

```
private int j=3;
private boolean[] a = new boolean[10];
private String s = "hi";
private String[] names = {"New York", "Boston", "Chicago"};
```

16.3 Final Variables

It is sometimes useful to have names that are associated with constant values. Here's an example:

```
private static final int MAXIMUM_SIZE = 100;
```

Here's another example, which appears in the built-in class `Math`:

```
public static final double PI = 3.141592653589793;
```

The keyword `final` forces the name to be permanently associated with the value instead of with a genuine variable.

The significance of the declaration of `PI` is that we can write `Math.PI` whenever we want the value of π. The ability to declare more mundane constants is important too. Suppose, for example, that the constant 100, intended as a maximum size, appears multiple times in a program. If we name the constant in the way shown above, we can then use the name rather than the value throughout the program. This 1) increases legibility, and 2) makes it easy to change the constant if need arises.

By convention, constants use names containing upper-case characters. Underscore characters are used to separate words if the constant name is based on a phrase, for example in `MAXIMUM_SIZE`.

16.4 Constructors

Classes are permitted to have multiple constructors, as long as each constructor has a distinct list of parameter types. Recall that class `StudentList` (Figure 15.4) had a constructor that took an int parameter that specified a maximum number of students that could be part of the list:

```
public StudentList(int size) {
    list = new Student[size];
    capacity = size;
    count = 0;
}
```

We could add a second constructor that takes no parameters:

```
public StudentList() {
    list = new Student[MAXIMUM_SIZE];
    capacity = MAXIMUM_SIZE;
    count = 0;
}
```

This would permit us to create a `StudentList` by writing either `new StudentList(100)` or `new StudentList()`.

If a class has no constructors, there is always a default no-argument constructor that does nothing. So if class `A` had no constructor, it would be fine to write `new A()`. On the other hand, the presence of any constructor eliminates our ability to use the default constructor. Given the existence of the one-argument constructor for `StudentList`, we can not use the no-argument constructor unless someone has explicitly written it.

Here's a better way to write the no-argument constructor for `StudentList`:

```
public StudentList() {
    this(MAXIMUM_SIZE);
}
```

The call to `this` means "call another constructor for this same kind of object." This type of call can only be made in a constructor, and it must appear as the very first statement. It's possible to put other code in the constructor after the call to `this`. (Please note, this use of `this` is totally unrelated to the use of `this` as a variable that refers to the current object.)

Constructors are almost always public. If a constructor is marked `private`, it can only be called from within the class. Sometimes a class will have one or more public constructors along with one or more intended for internal use.

The class `Math` provides an interesting illustration of the effect of the `public` and `private` keywords on constructors. `Math` exists only to hold static methods, such as `cos` and `sqrt`, and static variables, such as `PI`. No one is ever supposed to create a `Math` object. To prevent this possibility, `Math` contains a single, private, no-argument constructor that does nothing. This means that a call to `new Math()` will be interpreted as a call to a private constructor and will cause a compilation error.

16.5 Testing Objects for Equality

Suppose you have created a class `MyClass` and wish to test whether two `MyClass` objects are equivalent. It is usually not correct to write

```
if (a == b) ...
```

The problem with this code is that it asks "Are pointers `a` and `b` equal?" In other words, it asks whether `a` and `b` refer to the same object. It's possible that the pointers refer to distinct but equivalent objects, in which case the comparison will yield the wrong answer.

Java classes often provide an `equals` method that compares objects rather than pointers. This is what permits us to compare two `String` objects by writing something like

```
if (s.equals(t)) ...
```

As mentioned earlier, it is vital use this way of comparing strings rather than `==`. We will defer discussion of writing an `equals` method to section 21.2.

It's OK, by the way, to compare two variables for an enumerated type, e.g. two `Suit` variables, with the `==` operator. The technical reason for this is that a single object is created for each suit, and that two variables refer to the same suit if and only if they point to the same object.

16.6 Garbage Collection

Once created, an object will remain in the computer's memory for as long as it is needed. They are handled in the same way as arrays (see section 13.5), and in reality an array is just a special kind of object. If an object is no longer accessible via variables and other objects, its memory becomes available for reuse via garbage collection.

Here's an example illustrating a case which which an object becomes garbage:

```
Student s = new Student ("Jim", 2011, 923);
Student t = new Student ("Anne", 2012, 1011);
t = s;
```

After the reference in `s` is copied into `t`, the object for Jim is unreachable and becomes garbage.

16.7 Frequently Asked Questions

Appendix C contains answers to some frequently asked questions about objects.

Chapter 17

Interfaces

17.1 Stacks

A *stack* is a way of organizing a collection of data that permits the data to accessed in a particular way. Here the operations that can be used with a stack:

- *push:* takes a new data item and puts it "on the top" of the stack
- *pop:* removes and returns the top item
- *isEmpty:* checks if the stack contains any data

An easy way to visualize a stack is to think of a pile of trays in a cafeteria. Imagine that each tray in the pile has a number—that's the data—written on it. A *push* operation takes a new tray, writes a number on it, and adds it to the pile. A *pop* operation removes the top tray from the pile and returns the number written on it. These rules imply that if you *push* the values 1, 2, and 3 (in that order) and then do three *pops*, you'll get the values 3, 2, and 1, in that order. The idea of a stack has many applications in computer science.

So how could we write Java code to implement a stack? It turns out that there are several approaches. We will explore one of them, an array-based implementation, in this chapter. Before creating the implementation, it is valuable to give a clear abstract description of the operations that we need to support. Figure 17.1 shows how we can this is Java.

An *interface* is similar to a class, with the following differences:

1. The keyword `interface` is used instead of `class`.
2. There are no static variables or methods.

```
public interface Stack {

    public void push (int i);
    public int  pop ();
    public boolean isEmpty();

}
```

Figure 17.1: An interface of a stack of integers.

3. There are no instance variables.

4. There are no constructors.

5. The bodies of the instance methods are missing!

The point of an interface is to specify the operations that must be part of a real implementation, without offering any information on how the implementation will work.

Figure 17.2 presents an array implementation of a stack. It is an ordinary class, with the words `implements Stack` given as part of the header line. It is a legal implementation because it provides each of the methods listed in `Stack`.

Class `AStack` uses two private instance variables, one to hold an array of integers, and one to keep track of how many items are currently in the stack. The value held in `a[0]` is considered to be the bottom-most element in the stack, and the value held in `a[count-1]` is the top-most. Exceptions are thrown if we try to add an element to a stack that's already full or if we try to remove an element from a stack that's empty.

It's possible to avoid the problem of an overfull stack by adding a method to resize the array:

```
private void resize() {
    int[] b = new int[a.length*2];
    for (int i=0; i<a.length; ++i)
        b[i] = a[i];
    a = b;
}
```

This allocates an array twice the length of the old one, copies the stack elements to it, and makes instance variable `a` point to the new array. No pointers are kept to the old array, so it becomes available for garbage collection. If we have written

```
public class AStack implements Stack {

    private int[] a;
    private int count;

    public AStack(int size) {
        a = new int[size];
    }

    public AStack() {
        this(10);
    }

    public void push(int i) {
        if (count == a.length)
            throw new RuntimeException ("Array is full");
        a[count++] = i;
    }

    public int pop() {
        if (count == 0)
            throw new RuntimeException ("Stack is empty!");
        return a[--count];
    }

    public boolean isEmpty() {
        return count == 0;
    }
}
```

Figure 17.2: An array implementation of a stack.

method `resize()`, we can call it instead of throwing an exception when the array is full.

Once you have both an interface and an implementation, you can use them in a remarkable way. The following code illustrates the creation and use of an `AStack` object:

```
Scanner s = new Scanner(new FileReader(infile));
Stack stack = new AStack();

while (s.hasNextInt())
    stack.push(s.nextInt());

while (!stack.isEmpty())
    System.out.println(stack.pop());
```

This causes sequence of ints that are read to be printed in reverse order. The most interesting feature of this code is that we say `new AStack()` and then store the value in a variable of type `Stack`. This is OK because an `AStack` is, in effect, a kind of `Stack`.

Part of the value of using interfaces is that we are able to separate the specification of something from its implementation. We'll see a much different way to implement `Stack` in section 22.2.

17.2 An Interface for a Player Object

If you are implementing a game that involves human and/or computer players, you can use an interface to simplify your code. Figure 17.3 shows part of a program for a game that involves a board, tiles, and a set of players. There are many things that we've left undefined in the program, including classes `DrawPile` and `Board` and methods `getPlayerCount()`, `answerIsYes()` and `getName()`. The `Player` interface would be written in the following way:

```
public interface Player {
    public void drawTiles (Pile p);     // draw initial set of tiles
    public void play(Board b, Pile p);  // take a turn
    public boolean isDone();    // test if player has run out of tiles
    public int getScore();              // get the player's score
    public String getName();            // get the player's name
}
```

Classes `HumanPlayer` and `ComputerPlayer` would be implementations of `Player` that differ primarily in their `play` methods. In `HumanPlayer`, the `play` method

```
1   public class Game {
2
3       public static void main (String[] args) {
4
5           int n = getPlayerCount();
6
7           DrawPile pile  = new DrawPile();
8           Board    board = new Board();
9
10          Player[] players = new Player[n];
11
12          for (int i=0; i<n; ++i) {
13              System.out.println ("Is Player " + i + " a human?");
14
15              if (answerIsYes()) {
16                  String name = getName(i);
17                  players[i] = new HumanPlayer(i);
18              }
19              else {
20                  String name = getName(i);
21                  players[i] = new ComputerPlayer(i);
22              }
23              players[i].drawTiles(pile);
24          }
25
26          int currPlayerNum = 0;
27
28          while (true) {
29
30              Player p = players[currPlayerNum];
31              p.play(board, pile);
32              if (p.isDone()) break;
33              currPlayerNum = (currPlayerNum + 1) % n;
34          }
35
36          Player winner = null;
37          int bestScore = Integer.MIN_VALUE;
38
39          for (Player p : players) {
40              int pscore = p.getScore();
41
42              if (pscore > bestScore) {
43                  winner = p;
44                  bestScore = pscore;
45              }
46          }
47
48          System.out.println ("The winner is " + winner.getName());
49      }
50  }
```

Figure 17.3: A program for a game

would interact with the player to ask what move to make, while in `ComputerPlayer` it would make a decision automatically. The most critical line in Figure 17.3 is line 31:

```
p.play(board, pile);
```

which causes the appropriate `play` method to run, based on what kind of `Player` object is referred to by variable `p`.

You should note that while it is not legal to write `new Player()`, it is legal to write `new Player[]` on line 10 in order to create an array of `Player` reference variables. The constant `Integer.MIN_VALUE` is used on line 37 to initialize `bestScore` to an extremely negative value. See section 21.1 for more information on `MIN_VALUE` and other constants.

Chapter 18

Generic Types and Wrapper Classes

In Chapter 17, we explored how we could specify and implement a stack of integers. Suppose that we now need a stack of strings. Can we avoid duplicating all the code and creating a slightly modified copy? The answer is "yes."

Here is a revised version of the `Stack` interface:

```
public interface Stack {

    public void push (Object i);
    public Object pop ();
    public boolean isEmpty();
}
```

We could modify class `AStack` (Figure 17.2), replacing `int` with `Object` throughout, to obtain an implementation that is compatible with this interface.

The new stack could be used in the following way:

```
Stack s = new AStack();
s.push("This");
s.push("is");
s.push("fun");

while (!s.isEmpty())
    String t = (String) s.pop();
    ...
}
```

`Object` is the name of a class that can represent any kind of object. A string is an object, so it's possible to write `push(t)` for any string `t`. The unpleasant feature of this code is the necessity of writing doing a cast when a string is popped from the stack. We need to do the cast so that the generic object returned by the `pop` is recognized as a string.

Java uses a mechanism, *generic types*, that makes it relatively straightforward to create and use general-purpose classes. This is our final version of the `Stack` interface:

```
public interface Stack<E> {

    public void push (E i);
    public E pop ();
    public boolean isEmpty();
}
```

The final version of `AStack` appears in Figure 18.1. This version of the stack can be used as follows:

```
Stack<String> s = new AStack<String>();
s.push("This");
s.push("is");
s.push("fun");

while (!s.isEmpty())
    String t = s.pop();
    ...
}
```

Let's see how this works. The name given in angle brackets, in this case `E`, represents any class name. To create an `AStack` for strings, we write

```
new AStack<String>();
```

Variable `s`, which holds a pointer to the stack, is declared to as a `Stack<String>`. The code in `Stack` and `AStack` is generally straightforward, with the exception of lines 7 and 15 in `AStack` (Figure 18.1). We would like to write

```
a = new E[size];
```

but for subtle technical reasons this is not allowed. So we can finesse the problem by creating an array of `Object` and then casting the array to have type `E[]`.

```
1   public class AStack<E> implements Stack<E> {
2
3       private E[] a;
4       private int count;
5
6       public AStack(int size) {
7           a = (E[]) new Object[size];
8       }
9
10      public AStack() {
11          this(10);
12      }
13
14      private void resize() {
15          E[] b = (E[]) new Object[a.length*2];
16          for (int i=0; i<a.length; ++i)
17              b[i] = a[i];
18          a = b;
19      }
20
21      public void push(E i) {
22          if (count == a.length) resize();
23          a[count++] = i;
24      }
25
26      public E pop() {
27          if (count == 0)
28              throw new RuntimeException ("Stack is empty!");
29          return a[--count];
30      }
31
32      public boolean isEmpty() {
33          return count == 0;
34      }
35  }
```

Figure 18.1: An array implementation of a stack using generic types.

You should note that compiling `AStack.java` causes the following warning to appear:

```
Note: AStack.java uses unchecked or unsafe operations.
```

The warning doesn't mean that the compilation failed, just that there was an issue that you should know about. It's never good to ignore warnings, but in this case there is not a real problem. The warning can be eliminated by adding the line

```
@SuppressWarnings("unchecked")
```

immediately before the no-args constructor and before method `resize()`.

The need to deal with the (relatively few) complications that arise when writing generic code should not obscure the benefit of using generic types. Users of generic types are able to avoid doing frequent casts, which can greatly simplify programs. Many classes in the Java API are now based on generics and are much easier to use than their predecessors.

18.1 Wrapper Classes

Suppose that you want to have a stack of ints. It would be great if you could write

```
Stack<int> intStack = new Stack<int>();      // not legal
```

but this is not allowed because generics only work with objects. It is fine, however, to write this:

```
Stack<Integer> intStack = new Stack<Integer>();
```

`Integer` is an example of a *wrapper class*, a class that permits you to wrap a primitive value within an object so that you can use it somewhere where an object is expected. Other wrapper classes include `Double`, `Boolean`, and `Character`.

To wrap an int as an object, you simply construct an `Integer` object. So to push an int onto `intStack`, you can write something like

```
intStack.push(new Integer(i));
```

This would take the value of `i`, wrap it in an object, and then push it onto the stack. To remove the value from the stack, you can write

```
int j = intStack.pop().intValue();
```

Method `intValue()` extracts the int value from an `Integer` object. The extraction methods for `Double`, `Boolean`, and `Character` are `doubleValue()`, `booleanValue()`, and `charValue()` respectively.

If you simply wanted to pop and print a value, you could do it with

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

This works because the `toString()` method in `Integer` just returns a string version of the wrapped value.

18.2 Autoboxing and Autounboxing

Recent versions of Java have made it even easier to use primitive values where objects are expected. The following sequence of instructions is now legal:

```
Stack<Integer> intStack = new Stack<Integer>();
intStack.push(3);
int j = intStack.pop() + 2;
```

Variable `j` would be assigned the value 5.

Autoboxing is the automatic wrapping of a primitive value into an object from the corresponding wrapper class, and it occurs in any situation in which the primitive type can't be used but the wrapper class can.

Autounboxing is the automatic unwrapping of a wrapper object. It occurs anytime that the unwrapping produces legal code. Here are some examples showing when unwrapping would and would not occur.

```
Integer iObj = intStack.pop();                  // no unwrapping
int k = iObj * 7;                               // unwrapping occurs
System.out.println (iObj + 2);                  // unwrapping occurs
System.out.println (iObj + iObj);               // unwrapping occurs
System.out.println (iObj.toString().length())   // no unwrapping
```

Note that there is ambiguity in the meaning of the `+` operator in the third and fourth lines. It could mean concatenation of strings, meaning that each object reference would have an implicit `.toString()` appended, or it could mean addition of ints, meaning that autounboxing occurs. The ambiguity is resolved in favor of the second option.

Autoboxing and autounboxing work similarly for the other wrapper classes.

Chapter 19

Inheritance

Java permits programmers to design new classes that are derived from existing classes. Each new *subclass* can inherit methods and variables from its *superclass*. *Inheritance* makes it possible to reuse existing code easily and is one of the most powerful aspects of Java and other object-oriented languages.

Let's begin with an example. Consider the following class:

```
public class Shape {

    public int sides;

    public String toString() {
        return "This is a shape";
    }

    public int getSides() {
        return sides;
    }
}
```

In some method, perhaps the main method, you could write:

```
Shape s = new Shape();
System.out.println (s);
System.out.println (s.getSides());
```

The `println` statements would work in the expected way, producing:

```
This is a shape
0
```

Recall that all numeric instance variables have a default value of zero. That's the reason that `getSides()` returns 0.

Now consider a class that "extends" the idea of a shape:

```
public class Triangle extends Shape {

    public Triangle() {
        sides = 3;
    }

    public String toString() {
        return "This is a triangle";
    }

    public String hi() {
        return "Hello!";
    }
}
```

Here are some statements that might appear in the main method:

```
Triangle t = new Triangle();
System.out.println (t.getSides());
System.out.println (t);
System.out.println (t.hi());
```

These statements lead to the following output:

```
3
This is a triangle
Hello!
```

When we say `Triangle extends Shape`, we are declaring `Triangle` to be a subclass of `Shape`. A `Triangle` object inherits all the methods and variables in class `Shape`, and it *overrides* one of the methods. The reference to variable `sides` in the `Triangle` constructor works because `sides` is a variable in `Shape`. (We'll talk about privacy considerations later in section 20.3.) `Triangle` has no method `getSides()`, so it inherits the method from `Shape`. It does have a `toString()` method, which overrides the one in `Shape`. It also has a method `hi()`, so the reference to `t.hi()` works.

Now consider these statements:

```
Triange t = new Triangle();
Shape s = t;
System.out.println (s.getSides());
System.out.println (s);
System.out.println (s.hi());       // an incorrect statement
```

The assignment of `t` to variable `s` works because every `Triangle` is a legitimate `Shape`. The first `println` prints 3 because that is the value in the `Shape` object.

The second `println` prints "This is a triangle." This may seem surprising. Variable `s` refers to an object that is really a `Triangle`. It was created as a `Triangle`, and it maintains that identity even though we are now holding the reference in a `Shape` reference variable. So here's the rule: *when you apply a method to an object, the method that is used is chosen based on what the object really is.* In this case the object is really a `Triangle`, so the `toString()` method used is the one from `Triangle.`

The third `println` statement is illegal and produces a compilation error. In determining the correctness of the method call `s.hi()`, the compiler isn't smart enough to recognize that `s` holds a reference to a `Triangle` object. This fact is obvious during execution, but not during compilation. Class `Shape` doesn't contain a method called `hi()`, so the compiler does not permit the method to be applied to a `Shape` object.

The ability to execute different methods based on an object's "real type" is called *polymorphism* and is valuable in many settings. Here's an example. The Java graphics library defines a generic graphical object called a `JComponent`. There are many subclasses of `JComponent`, such as `JButton`, `JScrollBar`, and `JTextField`, each of which corresponds to a kind of graphical object that someone might want to use. Each of these classes, including `JComponent` has a method called `paintComponent()` for displaying itself in a graphics window.

In addition to classes describing individual graphical components, there's another class called `JPanel` that contains variables describing the location, dimensions, visibility, etc. of a window. It also contains an array of `JComponent` references that point to the various things that appear in the window. Now consider the code that is needed to display a window. After creating the frame of the window, it needs to display the components in the window. It can achieve this by simply invoking the `paintComponent()` method on each `JComponent` in the array. Each `JComponent` will be displayed based on what it really is (e.g. a `JScrollBar` or a `JButton`). Inheritance makes it easy to define the different kinds of components and polymorphism makes it very easy to write code to redraw a collection of components without worrying about what they actually are.

19.1 Class `Object`

A single class may have many subclasses. This was certainly the case with `JComponent`, and it occurs frequently in the Java API. Each class may extend only one other class, i.e. it can have only one superclass.

A subclass can itself have subclasses. For example, `B` might extend `A` and `C` might extend `B`. This would mean that `C` would indirectly extend `A`. Class `C` would have access to each method and variable declared in class `A`, unless `B` contained a declaration overriding the one in `A`.

If a class is not labelled as extending another one, it is implicitly a subclass of `Object`, a class defined in the Java API. `Object` is the ultimate superclass, in the sense that every class is a subclass of it, either directly or undirectly.

Class `Object` contains a collection of methods, some of which we will see in the pages ahead. The presence of these methods as part of `Object` means that they can be applied to any objects at all. It is often desirable to override these methods in the classes we write.

One method in `Object` is `toString()`. When called on an object of type `MyClass`, it produces a string of this form:

```
MyClass@47b480
```

This string gives the name of the class and the address (in hexadecimal notation) of the object. Not too interesting, right? The version of `toString()` in `Object` is effectively the default that's used when no other version is available. If `MyClass` had contained its own version of `toString()`, then any object of type `MyClass` would be printed with the new version. The new version would be also be used by subclasses of `MyClass` if they didn't have their own versions.

Because `Object` is a full-fledged class, it's possible to write something like

```
Object o = "hello there";
```

This copies a reference to a string into variable `o`. While there isn't much point to writing that particular line, it's often common to have code that passes a pointer to some particular kind of object into a method that is seeking a parameter of type `Object`.

19.2 Casts to Subclasses

It is possible to test if an object has a particular type and to cast reference variables from one type to another.

Suppose we have an `Object` reference variable called `o`. We can test if it refers to a `String` object by writing

```
if (o instanceof String) ...
```

The keyword `instanceof` is a binary operator that yields `true` if `o` is a `String`. Any object from subclass of `String` is, in every sense, a `String` and would yield `true`.

If `o` refers to a `String`, you can write

```
String s = (String)o;
```

The cast will succeed as long as `o` really is of type `String`; if not, a `ClassCastException` will occur. Doing the cast makes it possible, for instance, to write `s.length()`. We can't write `o.length()`, because `length()` is not a method in class `Object`.

19.3 Exceptions

In Java, exceptions are represented by objects. We've seen many kinds of exceptions, such as `ArrayIndexOutOfBoundsException`, `NullPointerException`, and so on, each of which is a distinct class in the Java API. We can write new class descriptions to define new kinds of exceptions.

Here's a simple example:

```
public class StackEmptyException extends RuntimeException {
    public StackEmptyException() {
        super("Stack is empty.");
    }
}
```

You can throw this kind of exception by writing

```
throw new StackEmptyException();
```

If a new exception class extends `RuntimeException`, it is an unchecked exception; if it extends `Exception`, it is a checked exception. (See section 11.4 for more information on the difference between these kinds of exceptions.)

The call to `super()` in the constructor of `StackEmptyException` invokes a constructor in the superclass, `RuntimeException`, which in this case records a message describing the exception. It's possible that the subclass constructor might do more work after the superclass constructor returns, but it does not do so in this example.

There is an important general rule: *all constructors in a subclass must, directly or indirectly, invoke a constructor in the superclass.* There are several possibilities:

- *A constructor can begin with a call to* `this(...)`. In other words, it invokes some other constructor in the same class.
- *A constructor can begin with a call to* `super(...)`. This causes it to invoke a constructor in the superclass, thereby ensuring that the superclass's part of the object is well-constructed.
- *Neither of the above occurs.* This causes an implicit call to the no-arguments constructor of the superclass. The superclass needs to have either an explicit no-arguments constructor or no constructors at all.

Because exception objects are full-fledged objects, it is possible at add instance variables and methods to them. You could, for example, use

```
throw new MyException(3,4,5);
```

to throw a new exception object that stores various values that provide information about what went wrong. If you catch the exception with

```
catch(MyException e) { ... }
```

then you can use variable `e` to access the information that's embedded in the exception object.

19.4 Abstract Classes

An *abstract class* is a class in which one or more methods are not implemented. Instead of writing

```
public int f() { ... }
```

where the code in the ellipsis does something or other, we might write

```
public abstract int f();
```

If some method is abstract, the class itself must be labelled as abstract, for example with

```
public abstract class MyClass {
```

If `MyClass` is abstract, it is illegal to write

```
MyClass c = new MyClass();
```

This illegality is appropriate, because it's hard to know what it would mean if we tried to run `c.f()` if we had never provided code for `f()`.

Suppose, on the hand, we had a subclass `MySubclass` that provided the missing method. We could then write

```
MyClass c = new MySubclass();
```

and the meaning of `c.f()` would be clear.

The general rule is that objects can be created based on subclasses of an abstract class as long as all of the missing methods are provided. If a subclass does not provide all of the missing methods, it must itself be labelled as abstract.

Abstract methods are useful in many applications. A typical example concerns the graphical components mentioned on page 161. `JComponent` is an abstract class, because it doesn't make sense to have a `paintComponent()` method for a general graphical component. What would it do? On the other hand, we need to force subclasses of `JComponent` to provide the method.

19.5 Relationship to Interfaces

An interface is effectively an abstract class that only contains abstract instance methods. Abstract classes can contain static variables and methods, constructors, and variables, along with implementations of some methods.

Interfaces do provide extra power by offering a way to get around Java's requirement that a class have only one superclass. We could write, for example,

```
public class A extends B implements C, D, E {
```

This lets `A` be a subclass of `B` and simultaneously forces it to include the methods listed in interfaces `C`, `D`, and `E`.

Languages such as C++ use *multiple inheritance* and permit multiple superclasses. This causes a number of complications that have been avoided in Java.

19.6 Access to Variables

While subclasses inherit access to the variables of superclasses in a straightforward manner, it is important to note that variables are handled differently than method calls. Suppose we had these declarations:

```
class A {
    public int i;
    public int j:
```

```
    public int f() {...}
}
class B extends A {
    public int i;
    public int k;
    public int f() {...}
}
```

Now suppose we have the lines

```
B b = new B();
b.i = 3;
b.j = 4;
b.k = 5;
A a = b;
```

It is a compilation error to refer to `a.k`, because `k` is not a known instance variable in `A`. Accessing `a.j` yields the value 4. Accessing `a.i` yields the value 0, which isn't necessarily what one would expect.

Here's what's happening. Objects from a class contain separate copies of all of the variables declared in the class and all of it superclasses. So class `B` has its `B` variables, its `A` variables, and its `Object` variables. When the pointer to the `B` object is stored in variable `a`, the `B` variables become inaccessible. So the access to `a.i` yields 0, the initial value stored in the `A` part of the object. (The handling of method calls is different, and it's important to recall that accessing `a.f()` calls the method in class `B`.)

We would regain access to the `B` part of the object by simply using variable `b`. If that variable were no longer available, we could write

```
B bb = (B)a
```

to cast the `A` reference back to a `B` variable.

Chapter 20

Packages

A *package* is a collection of related classes and interfaces. We've used three packages so far:

- `java.lang`: The classes and interfaces in this package are closely associated with the Java language itself. We've worked with these classes so far: `String`, `Object`, `Math`, `System`, `Integer`, `Character`, `Double`, `Boolean`, `Exception`, `RuntimeException`, and numerous exception classes.

- `java.util`: This package contains utility classes and interfaces. We've used one so far, `Scanner`, and you'll learn about many more in Chapter 21.

- `java.io`: This package contains classes and interfaces associated with input and output. We've used four: `PrintWriter`, `StringWriter`, `FileReader`, and `FileNotFoundException`.

The full name of a class or interface always includes its package name, so the full name for `String` is `java.lang.String`.

For simplicity, the word "class" will mean "class or interface" throughout this chapter. There is no difference in the way that classes and interfaces are handled in packages.

20.1 Using Packages

Using a class from a package involves three steps: 1) making sure that the package is installed on your system, 2) making sure that you issue commands such as `javac` and `java` in a way that lets you access it, and 3) referring correctly to the class in your programs.

For packages distributed with the Java system, most of which start with the prefix `java.` or `javax.`, steps 1 and 2 are easy; the packages are automatically installed correctly and no special action is needed when compiling or running. To install packages obtained in some other way, consult the documentation or a local expert.

Within your programs, you can always refer to a class by using its long name, e.g. `java.lang.String` or `java.io.FileReader`. It's always fine to omit the package name from classes in `java.lang`. If you want to use the short form of a name from another package, use an `import` line at the very top of your files. For example, if you write

```
import java.io.FileReader;
```

at the top of a file, you can write `FileReader` instead of `java.io.FileReader` throughout the file. You can also write something like

```
import java.io.*;
```

to import all the classes from a given package. This form should be avoided unless you have a good understanding of the entire package and are able to anticipate conflicts between names in the package and other names that you want to use.

20.2 Creating a Package

It is useful to create a package for a large project or a portion of a project or if you have a collection of classes that you'd like to distribute.

On a Unix system, you can develop a package in the following way. Suppose you wanted to create a package called `amherst.cs12`. You would need to have three directories:

- A top-level directory, either your home directory or another directory of your choice. For simplicity, let's assume that you have a directory called `projects`.
- A subdirectory of `projects` called `amherst`.
- A subdirectory of `amherst` called `cs12`.

All of your classes should appear in `cs12`, and each should have the following line at the very top:

```
package amherst.cs12;
```

Classes in the package never need to use the `amherst.cs12` prefix when referring to each other. In other words, there's always an implicit import of other classes in the same package.

To compile your classes, issue the command

```
javac amherst/cs12/*.java
```

while you are directory `projects`.

To execute your program, issue the command

```
java amherst.cs12.Main
```

while you are in `projects`, assuming that `Main` is a class containing a main method.

You can create a `jar` file, an archive file containing the whole package by running

```
jar cf cs12.jar amherst/cs12/*.class
```

while in `projects`. The resulting `jar` file can be moved to other directories, sent to others, etc. If someone is in any directory that has a copy of `cs12.jar`, `Main`'s main can be executed by running

```
java -cp cs12.jar amherst.cs12.Main
```

More generally the `-cp` option can be used whenever you run `java` or `javac` in order to gain access to the classes in the `jar` file. For more information, consult the documentation or a local expert.

A package name can have any number of levels. The example in this section, `amherst.cs12`, used two, but it's possible to use only one level or more than two.

20.3 Privacy Levels

Now that we know about inheritance and packages, it is possible to understand exactly how privacy levels work. Variables and methods can be labelled as `public`, `private`, `protected`, or none of the above:

- `public`: A public variable or method can be accessed in any class in any package.
- `private`: A private variable or method in class `A` can be accessed only in `A`.
- `protected`: A protected variable or method in class `A` can be accessed in any class that is 1) in the same package, or 2) is a subclass of `A`. In other words, it's fine to create a subclass of `A` in some other package, and that subclass will have access to `A`'s protected variables.

- none of the above: A variable or method without a designated privacy level has *package-level* privacy. This would occur, for example, if we wrote

  ```
  int count;
  ```

  instead of

  ```
  private int count;
  ```

  when we declared an instance variable. In this case, the variable or method would be accessible in any class in the same package.

There is currently no privacy level permitting access in subclasses but not throughout the same package.

20.4 Public and Nonpublic Classes

Classes can have either public or package access. In other words, it is possible to write either

```
public class MyClass {
```

or

```
class MyClass {
```

Classes that are not public can only be referred to by other classes in the same package. Furthermore, if you want to execute a main class, that class must be public.

20.5 The Unnamed Package

If you omit the package designation in a collection of Java files, they are considered to be in the *unnamed package*. It is routine in beginning programming classes to create small programs without naming a package, and indeed this is what lets us compile and run with simple commands such as

```
javac *.java
```

and

```
java Main
```

As programs become more complex, the advantages of using packages to organize your classes become more significant.

Chapter 21

The Java API

The Java API (Application Programming Interface) is a vast collection of classes and interfaces providing many facilities useful to Java programmers. In this chapter we will explore parts of some of these classes. Programmers should explore the API documentation, available on-line at

`http://download.oracle.com/javase/6/docs/api/`

for more information on all aspects of the API.

21.1 Constants

Numerous constants are defined in the API, including the following, all of which are in classes in the `java.lang` package:

- `System.out`: a `PrintWriter` associated with standard output from an application, which is usually directed to the user's terminal window.
- `Math.PI`: a double containing the value of π.
- `Math.E`: a double containing the value of e, the base of the natural logorithm.
- `Integer.MAX_VALUE`: an int containing the largest possible value, $2^{31} - 1$.
- `Integer.MIN_VALUE`: an int containing the smallest possible value, -2^{31}.
- `Double.MAX_VALUE`: a double containing the largest possible value, $(2 - 2^{-52}) \cdot 2^{1023}$.
- `Double.MIN_VALUE`: a double containing the smallest possible positive value, 2^{-1074}.

- `Double.NaN`: a double containing NaN, the "not a number" value.
- `Double.POSITIVE_INFINITY`: a double containing a representation of ∞.
- `Double.NEGATIVE_INFINITY`: a double containing a representation of $-\infty$.

21.2 java.lang.Object

`Object`, discussed in section 19.1, is the root class in Java, In addition to `toString()`, discussed earlier, there are two important methods for intermediate programmers.

The `equals` method tests equality of objects. The header line is

```
public boolean equals (Object o)
```

The implementation of this method in `Object` returns `true` if `o` refers to the same object as `this`. In other words, two reference variables refer to equivalent objects only if they point to the very same object.

It is often desirable to override the `equals` method in subclasses. Suppose a `Card` object contains this code:

```
public class Card {
    private int value;
    private Suit suit;      // Suit is an enumerated type
```

It would be appropriate to include the following equality-testing method:

```
    public boolean equals (Object o) {
        if (o instanceof Card) {
            Card c = (Card)o;
            return suit==c.suit && value==c.value;
        }
        else return false;
    }
```

We can correctly test equality of both suits and values with the `==` operator, but it is often appropriate to use calls to `equals` to test equality of subparts of an object when you write your own `equals` method.

It is unfortunately impossible to avoid doing the `instanceof` test and cast in writing an `equals` method. Programmers should be careful to avoid taking the shortcut of letting the parameter to this method have a type other than `Object`. Such a method would not provide a true override of the `equals` method in `Object` and can cause unexpected behavior if the object is used with other parts of the API.

If a class overrides the `equals` method, it should also override the `hashCode` method:

```
public int hashCode()
```

This method should return an int and should follow these rules:

- Calls to `hashCode` for two equal objects *must* return the same value.
- Calls to `hashCode` for two unequal objects should, if possible, return distinct values, but this is not an absolute requirement.

The point of hash codes is to permit efficient searching in the `HashSet` and `HashMap` classes and in any other classes that have been built to use them. If you aren't using `HashSet` or `HashMap`, you can often get away without providing a `hashCode` method.

When writing a `hashCode` method, it's important to try to return distinct values for unequal objects, because too many *collisions* can undermine the performance of a `HashSet` or `HashMap`. Here is a method that would be appropriate in class `Card`:

```
public int hashCode() {
    return suit.hashCode()*13 + value;
}
```

21.3 java.lang.Comparable<T>

`Comparable` is an interface that can be used for ordering a collection of elements. The interface lists one method:

```
public int compareTo(T o)
```

The method assumes that both `this` and `o` have values drawn from an ordered set. The method should return zero if `this` and `o` are equal, a positive number if `this` has a larger value than `o`, and a negative order if `this` has a smaller value.

Class `String` implements `Comparable<String>`. In other words, strings can be compared to strings, and the results that are produced will be based on lexicographic (alphabetic) order. A call to

```
"him".compareTo("her")
```

would produce a positive value because "him" is greater than "her," in the sense that it comes later in alphabetic order.

If a class implements `Comparable`, it is possible to make calls to `compareTo` in order to sort a collection of elements. This implies, of course, that calls to `compareTo` should yield consistent results, i.e. if `a.compareTo(b)` and `b.compareTo(c)` are both positive, then `a.compareTo(c)` must be positive.

You will often benefit from writing classes that implement `Comparable`. Certain classes in the API, such as `TreeSet`, must be used with classes of this kind.

21.4 java.lang.StringBuffer

A `StringBuffer` represents a modifiable string, which makes it much different than `String`. Consider the following code:

```
public static String convertToString (char[] a) {
    String s = "";
    for (char c : a)
        s = s + c;
    return s;
}
```

This method correctly constructs a string from an array of characters, but it is highly inefficient, because `String` objects are inherently unmodifiable. When we run `s = s + c;`, a whole new string is allocated in dynamic memory, and the old string is copied into it along with a new character. If `a` contains 10,000 characters, then 10,000 strings are created, and there are repeated copies of longer and longer strings.

The following is much more efficient:

```
public static String convertToString (char[] a) {
    StringBuffer b = new StringBuffer();
    for (char c : a)
        b.append(c);
    return b.toString();
}
}
```

A `StringBuffer` is inherently modifiable, and the `append` operation can be completed without required a whole string to be copied. The class contains many other methods, such as ones that permit particular positions in the string to be inspected or changed.

21.5 java.lang.Iterable<T>

`Iterable` is an interface. Suppose `o` is an object from a class that implements `Iterable<T>`. It is possible to access, one by one, a sequence of elements of type `T` from `o`. In particular, it is possible to write

```
for (T t : o) {
   // do something with element t
}
```

This is another example of the for-each loop that we saw in section 14.1. Effectively, an array of type `String[]` can be said to implement `Iterable<String>`, in the sense that we could write

```
for (String s : theArrayOfStrings) {
    // do something with s
}
```

We'll see lots of examples of `Iterable` classes in this chapter.

Technically, when a class implements `Iterable<T>`, it must provide a method with this header:

```
public Iterator<T> iterator()
```

In other words, it must be able to return an `Iterator`, which becomes the basis of the loop.

21.6 java.util.Iterator<T>

An object based on an implementation of `Iterator` permits one to iterate across the elements that are part of some underlying object. Three methods are specified in the interface:

```
public boolean hasNext();

public T next();

public void remove();
```

`hasNext()` tells whether there is another element available in the underlying object, `next()` returns it, and `remove()` removes whatever object was returned last from the underlying object. Programmers frequently decline to provide a real implementation `remove()`, in which case they are permitted to throw a `java.lang.UnsupportedOperationException`. If `next()` is called when `hasNext()` is false, a `NoSuchElementException()` is thrown.

`Scanner` is an example of class that implements `Iterator<String>`.

It is worth noting that this use of the for-each loop:

```
for (T t : o) {    // o is from a class implementing Iterable<T>
    // do something with element t
}
```

is in fact transformed into the following:

```
Iterator<T> it = o.iterator();
while (it.hasNext()) {
    T t = it.next();
    // do something with element t
}
```

21.7 java.util.Comparator<T>

A class that implements `Comparator<T>` makes it possible to determine the order of two elements of type `T`. In particular, it provides a method

```
public int compare (T a, T b)
```

It returns 0 if the objects are equal, a positive number if `a` is larger, and a negative number if `b` is larger.

`Comparator` is obviously similar to `Comparable`. Its value comes from the ability to specify an order outside the given class. For example, we could implement `Comparator<String>` in a new class and specify an ordering distinct from the usual lexicographic one.

21.8 java.util.Collection<E>

A class that implements `Collection<E>` holds a collection of elements of type `E`. In general, duplicate elements are permitted and no particular order is maintained, although implementations can adopt their own policies. `Collection<E>` extends `Iterable<E>`, so it is possible to iterate over elements in the collection.

Methods specified in the interface include:

```
boolean add(E o);   // adds an element
void clear();       // removes all elements
void remove(E o);   // removes one element equal to o
boolean isEmpty();  // Is the collection empty?
int size();         // How many elements are there?
```

21.9 java.util.List<E>

This is an subinterface of `Collection<E>`. A `List` holds an ordered sequence of elements. Methods include:

```
boolean add(E o);          // adds an element at the end
boolean add(int index, E o);  // adds in a particular position
boolean contains(E o);     // tests if an element is in the list
E get(int index);          // gets the element in a particular position
E remove(int index);       // removes the element in a particular position
```

In addition, of course, all of the methods in `Collection` are available in any implementation of `List`.

21.10 java.util.ArrayList<E>

This is an implementation of `List<E>` in which the elements are held in an array. The array is resized if necessary in order to accomodate new elements. When the array capacity is increased, it is done in a way that ensures that many elements can be added before another resizing is needed.

It is relatively expensive to add or remove new elements from the front part of a long `ArrayList`. On the other hand, it's very inexpensive to get the element at a particular index. These characteristics distinguish it from a `LinkedList`.

21.11 java.util.LinkedList<E>

This an implementation of `List<E>` based on a linked list, which is the topic of Chapter 22. In addition to the methods specified in `List`, it contains several others, including these:

```
void addFirst(E o);      // adds to beginning of list
void addLast(E o);       // adds to end of list
E    getFirst();         // returns first element
E    getLast();          // returns last element
E    removeFirst();      // removes first element
E    removeLast();       // removes last element
ListIterator<E> listIterator(index);   // returns a list iterator
```

None of the first six methods adds new functionality, because each can be using the `add`, `get`, or `remove` method from `List`. The seventh yields a `ListIterator`, essentially an `Iterator` that can move both forward and backward within the list.

The performance characteristics of linked lists are much different than arrays. Changes or lookups at either end of the list are inexpensive. In general, getting an element by index or making a change in the middle of the list is expensive. Accessing elements or making change near the position of a `ListIterator` is inexpensive.

21.12 java.util.Set<E>

A `Set<E>` is a subinterface of `Collection<E>` in which all elements have distinct values.

21.13 java.util.TreeSet<E>

A `TreeSet<E>` is an implementation of `Set<E>` in which the elements are stored based on an order. The order can either be the natural order for `E`, assuming that `E` implements `Comparable<E>`, or it can be obtained using a `Comparator<E>` object that is specified when the `TreeSet` is constructed. If you iterate over a `TreeSet`, either by using an `Iterator` or (equivalently) with a for-each loop, you obtain the elements in sorted order.

21.14 java.util.HashSet<E>

A `HashSet<E>` is an implementation of `Set<E>` that is organized very differently from `TreeSet<E>`. It is designed so that `add`, `remove`, and `contains` are all very efficient, even if the `HashSet` contains huge numbers of elements. Using an iterator for a `HashSet` will yield the elements in no particular order.

An important note: class `E` must have a correct `hashCode` method if it has overridden the `equals` method.

21.15 java.util.Map<K,V>

An implementation of `Map<K,V>` gives a way to look up the value (of type `V`) that's associated with a particular key (of type `K`). All keys must be distinct. Methods that must be implemented include:

```
V put(K key, V value);     // puts a (K,V) pair into the map
V get(K key);              // gets the value for a particular key
V remove(K key);           // removes a key and its value
boolean containsKey(K key);   // checks if a key is present
Set<K> keySet();           // returns the set of keys
```

```
Collection<V> values();    // returns the collection of values
int size();                // returns the number of (K,V) pairs in the map
```

21.16 java.util.HashMap<K,V>

HashMap<K,V> is an implementation of Map<K,V>. The get, put, remove, and containsKey methods are all very efficient, even if the map contains huge numbers of key-value pairs.

As with HashSet, class K must have a correct hashCode method if it has overridden the equals method.

Chapter 22

Linked Lists

A *linked list* is a way of storing a sequence of data by stringing together a number of small objects, which we will call *cells*. Consider the following class:

```
public class IntCell {
    public int value;
    public IntCell next;

    public IntCell (int v, IntCell n) {
        value = v;
        next = n;
    }

    public String toString() {
        return "Value is " + value;
    }
}
```

Note that this declaration says that an `IntCell` contains an `IntCell`. That isn't a contradiction, because `next` is a reference to an `IntCell`, not an `IntCell` object. So an `IntCell` contains two instance variables, an int and a reference.

We could represent a list with one element in it by writing

```
IntCell one = new IntCell(6, null);
```

Variable `one` points to a cell that contains the value 6. The `next` field of the cell is null, which indicates the end of the list.

We could represent a list with two elements in it by writing

```
IntCell two = new IntCell(4, one);
```

Variable `two` points to a cell with value 4. The next field of that cell points to a cell with value 6. The next field of the second cell points to null. We conclude that the list is [4,6].

We could represent a list with three elements in it by writing

```
IntCell three = new IntCell(5, two);
```

And so on. For convenience, we can even say that

```
IntCell zero = null;
```

represents a list with no elements.

Here are some `println` statements that we might use based on variable `three`, along with the output that is produced:

```
println (three);                        // prints "Value is 5"
println (three.value);                  // prints "5"
println (three.next);                   // prints "Value is 4"
println (three.next.value);             // prints "4"
println (three.next.next);              // prints "Value is 6"
println (three.next.next.value);        // prints "6"
println (three.next.next.next);         // prints null
println (three.next.next.next.value);   // NullPointerException
```

22.1 Using Loops and Recursion with Linked Lists

It isn't necessary to have a distinct local variable for each cell in a linked list. Here, for example, is a method that can construct a list of arbitrary length by reading values from a `Scanner`:

```
public static IntCell buildList(Scanner s) {
    IntCell list = null;
    while (s.hasNextInt())
        list = new IntCell(s.nextInt(), list);
    return list;
}
```

If the values read from the `Scanner` were 1, 2, and 3, in that order, then the linked list would have 3 at the front, followed by 2, and then followed by 1.

A linked list can be printed (or processed in some other way) either by using a loop or by using recursion. Here's a loop for printing a linked list:

```
public static void printList (IntCell p) {
    while (p != null) {
        System.out.println(p.value);
        p = p.next;
    }
}
```

The method works by moving variable p down the list, so that it points to each cell in turn. It's important to remember that changing variable p within this method doesn't affect the calling method at all.

Here's a recursive method for the same task:

```
public static void printList2 (IntCell p) {
    if (p != null) {
        System.out.println(p.value);
        printList2(p.next);
    }
}
```

We can process linked lists in many other ways. Here's a method to find the maximum value in a linked list:

```
int findMax (IntCell p) {

    if (p == null)
        throw new RuntimeException (
                    "Can't find max of an empty list!");
    else if (p.next == null)
        return p.value;
    else
        return Math.max (p.value, findMax(p.next));
}
```

Here's a method to test if a given value appears in a list:

```
boolean findValue (IntCell p, int value) {

    if (p == null)
        return false;
    else if (p.value == value)
        return true;
    else
        return findValue (p.next, value);
 }
```

Here's a method that determines the length of a linked list:

```
int listLength (IntCell p) {
    if (p == null)
        return 0;
    else
        return length(p.next) + 1;
}
```

Here's a method that copies the values in a linked list into an array:

```
int[] convertToArray (IntCell p) {

    int length = listLength(p);
    int[] a = new int[length];

    for (int i=0; i<length; ++i) {
        a[i] = p.value;
        p = p.next;
    }
    return a;
}
```

Here's a method that takes a linked list and reverses it, so the first cell becomes last, and so on. The method returns a reference to the new first cell in the list. This example's a good bit trickier than the ones above.

```
IntCell reverseList (IntCell p) {

    if (p == null) return null;
    else if (p.next == null) return p;
    else {
        IntCell newList = reverseList(p.next);
            // p.next now points to the last item in newList
        p.next.next = p;
            // the next field of p.next now points to p
        p.next = null;
            // p is now the last item in the list
        return newList;
    }
}
```

Here's a method to add a value to the end of a list. Again, it returns a reference to the beginning of the list:

```
IntCell addToEnd (IntCell p, int value) {

    if (p == null)
        return new IntCell (value, null);
    else {
        p.next = addToEnd (p.next, value);
        return p;
    }
}
```

Here's an iterative version of the same method:

```
IntCell addToEnd (IntCell p, int value) {

    if (p == null)
        return new IntCell (value, null);
    else {
        IntCell t=p;
        while (t.next != null)
            t = t.next;
        t.next = new IntCell (value, null);
        return p;
    }
}
```

22.2 A Stack Implementation Using a Linked List

A linked list can be used in an implementation of the `Stack` interface shown on page 153. The implementation, shown in figure 22.1, is an alternative to the array implementation in figure 18.1. This implementation has a number of interesting features:

- A cell is defined to carry a value of a generic type, generalizing the int-based cells used previously in this chapter.

- The `Cell` class is established as an *inner class*. The `Cell` class is private, so it can't be used outside the `LLStack` class. That's appropriate, because no other class needs to know what is going on in this implementation. The inner class can be used freely within `LLStack`, and it is permitted to use variables from the outer class, although that wasn't necessary here. It is common to set up helper classes as inner classes. `Cell` is a helper in this case because it exists only to support the work of `LLStack`.

- There is only one instance variable, a pointer to the top `Cell`.

- There are no constructors at all. The only initialization used sets `top` to be null. Even that could have been omitted, because null is the default value of a pointer.

```
1   public class LLStack<E> implements Stack<E> {
2
3       private class Cell {
4           E value;
5           Cell next;
6
7           Cell (E v, Cell n) {
8               value = v;
9               next = n;
10          }
11      }
12
13      private Cell top = null;
14
15      public void push(E i) {
16          top = new Cell(i, top);
17      }
18
19      public E pop() {
20          if (isEmpty())
21              throw new RuntimeException ("Stack is empty!");
22          E result = top.value;
23          top = top.next;
24          return result;
25      }
26
27      public boolean isEmpty() {
28          return top == null;
29      }
30  }
```

Figure 22.1: A linked list implementation of a stack.

22.3 Singly and Doubly Linked Lists

The linked lists discussed in this chapter have been *singly linked lists*, lists in which pointers go from each cell to the next in a single direction. It is possible to create a `doubly` linked list in which pointers go both ways. Doubly linked lists are used in the implementation of `java.util.LinkedList`. Without back pointers, it is difficult to make modifications in the middle of a list.

Doubly linked lists and other data structures will be explored further in future editions of this text.

Chapter 23

Graphical Applications

Now that we've covered many of the fundamentals of Java programming, we can consider the question of creating graphical programs. There are two kinds of graphical programs, applications that you can start at the command line or by clicking on an icon, and applets that can run within a browser. We'll start with applications and move to applets later.

Throughout this chapter, key ideas as labelled as **Graphics ideas**. There are a number of new techniques, and lots of little details, involved in creating graphical programs.

Graphics Idea 1 *Graphical applications are event-driven.*

There are two aspects to a graphical application, how it appears on the screen and how it responds to *events* such as mouse clicks and keyboard inputs. The structure of a graphical application is strikingly different than that of an ordinary application. In many ordinary applications, there is a main loop in which input is received and processed. Most graphical applications work differently and simply provide a collection of methods for handling different events as they occur. In effect, the user of an *event-driven* program provides the loop when he or she uses the mouse or keyboard to generate a sequence of events.

Graphics Idea 2 *A graphical application is based on a collection of components.*

Consider an application that creates and manages the window shown in Figure 23.1. The goal of this application is simple: to show how many times the button has been clicked. This program is available as an applet at

`http://www.cs.amherst.edu/lam/applets/Counter/counter.html`

The Java files are available in the directory

```
http://www.cs.amherst.edu/lam/applets/Counter
```

Figure 23.1: A simple graphical window

There are four graphical elements in the window, each of which is represented by a different kind of graphical component:

- A `JButton` represents the button.
- A `JLabel` represents the text that is displayed.
- A `JPanel` represents the overall "contents" of the window, i.e., the button, the text, and the space around them.
- A `JFrame` represents the entire window, including the close, minimize, and maximize buttons.

(Don't worry yet about the full distinction between the `JPanel` and the `JFrame`.) The application needs to handle several kinds of events: clicking of the four different buttons ("click me", minimize, maximize, and close) and resizing of the frame.

These four classes used here, `JButton`, `JLabel`, `JPanel`, and `JFrame`, are part of the Java Swing library. Swing works with the Java AWT (Abstract Window Toolkit) and provides a powerful mechanism for creating graphical programs. In order to access the Swing and AWT libraries, you should generally include these lines in the classes that you create:

```
import java.awt.*;
```

```
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
```

You should consult the API documentation extensively for details on the classes that we discuss in this chapter.

Figure 23.1 is, by the way, a screenshot from a Macintosh. On other systems there would be minor differences in the appearance of the frame, buttons, and fonts. One of the interesting features of Swing is the ability to change the look-and-feel so that the appearance of windows follows the rules of other systems.

Graphics Idea 3 *Most graphical programs are written by creating subclasses of classes in the Swing and AWT packages.*

Figure 23.2 displays the code for class `CounterPanel`, which is a subclass of `JPanel` and which does most of the work in the counter program.

Graphics Idea 4 *A* `JPanel` *is a graphical component that can contain other components.*

A `JPanel` can "contain" other components, in the sense that other components can be incorporated graphically and operationally into it. In Figure 23.2, a `JLabel` is created and added on line 16, and a `JButton` is created and added on line 23. (Recall that it's possible to do an assignment within an expression; that's how variables `label` and `button` are set.)

The subcomponents in a `JPanel` are arranged by a *layout manager*. Line 14 specifies that we want to use a `BoxLayout` in which the subcomponents appear in a single column. (If the second argument were `BoxLayout.X_AXIS`, it would be a single row.) Lines 19 and 25 ensure that the center lines of the two components are aligned (as opposed to their left or right edges). Line 21 places a 50-pixel gap between the subcomponents.

Lines 17 and 18 set the font size and color for the label, superseding the smaller black font that would otherwise be used.

Line 27 places a 50-pixel empty border on all four sides of the panel. Without this line, the panel would appear very cramped.

Graphics Idea 5 *A button click is an* `ActionEvent` *and is handled by an* `ActionListener`.

A class that implements `ActionListener` provides an `actionPerformed` method for processing `ActionEvent`s. In Figure 23.2, class `CounterPanel` provides such a method.

```
1   import javax.swing.*;
2   import java.awt.*;
3   import java.awt.event.*;
4   import javax.swing.border.*;
5
6   public class CounterPanel extends JPanel implements ActionListener {
7
8       private int        counter = 0;
9       private JLabel     label;
10      private JButton    button;
11
12      public CounterPanel() {
13
14          setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
15
16          add (label  = new JLabel(counter + ""));
17          label.setFont(new Font ("Serif", Font.BOLD, 48));
18          label.setForeground(Color.BLUE);
19          label.setAlignmentX(0.5f);
20
21          add (Box.createVerticalStrut(50));
22
23          add (button = new JButton("Click me!"));
24          button.addActionListener(this);
25          button.setAlignmentX(0.5f);
26          button.setFocusable(false);
27
28          setBorder(new EmptyBorder(50, 50, 50, 50));
29      }
30
31      public void actionPerformed (ActionEvent e) {
32
33          if (e.getSource().equals(button)) {
34              counter++;
35              label.setText(counter + "");
36          }
37      }
38  }
```

Figure 23.2: CounterPanel.java

```
1   import javax.swing.*;
2
3   public class CounterApp extends JFrame {
4
5       private CounterApp() {
6           super("Counter");
7           setDefaultCloseOperation(EXIT_ON_CLOSE);
8           setContentPane(new CounterPanel());
9           pack();
10          setVisible(true);
11      }
12
13      public static void main (String[] args) {
14          new CounterApp();
15      }
16  }
```

Figure 23.3: CounterApp.java

On line 24, the call `button.addActionListener(this)` specifies that `ActionEvent`s generated from clicking `button` should be referred to the `CounterPanel` itself. The `actionPerformed` method checks that the source of the event was `button` and then increments `counter`, the count of clicks. It resets the text for `label`, which causes the window to be updated.

Graphics Idea 6 *A* `JFrame` *is top-level graphical component.*

Figure 23.3 shows the main class for the counter program. `CounterApp` is a subclass of `JFrame`, in other words, it is our version of the overall window. The main method simply creates a `CounterApp` object and exits. The constructor for `CounterApp` makes a series of calls to methods inherited from `JFrame`. The `super` call on line 6 invokes the constructor in `JFrame` and sets the title of the frame. The call to `setDefaultCloseOperation` ensures that clicking the close button will cause the program to terminate. The call to `setContentPane` specifies that the contents of the window will be a `CounterPanel`. The `pack()` call uses the specifications of the various graphical subcomponents and builds the frame and its contents. The `setVisible(true)` call ensures that the window is visible, not invisible. Each of these calls, except the one that sets the title, is critical and should appear whenever you create a subclass of `JFrame`.

The minimize and maximize buttons and resizing of the top-level window are all handled automatically by Swing. Resizing is a delicate issue that we'll discuss

more soon.

In many cases, you can use the code in Figure 23.3 almost verbatim, changing only the names, in your `JFrame` class. Structuring your `JFrame` class in this way will make it easy to turn your application into an applet.

23.1 Threads

All of the non-graphical programs that we discussed in previous chapters used a single *thread of execution*. When execution begins, a main method is called and begins its work. Execution may involve branching, loops, and method calls, but there is always a single "current location" in the program.

Graphical programs are often organized differently. There are multiple threads of execution when the program is running. Essentially this means that there are multiple programs running simultaneously that share the same memory but handle different tasks.

The counter program involves two threads:

- The *main thread* is the one that starts first and begins by running the main method. It creates a `CounterApp` object and terminates. The creation of the first graphical object causes the other thread to start running.
- The *event thread* responds to events triggered by user actions, such a mouse clicks, mouse movements, and keystrokes. The event thread is also responsible for the display of components. It redisplays them as needed, for example, when the window is first created, when it is resized, or when text of a label changes.

The significance of the use of multiple threads will become more evident as we consider more complex graphical programs. In particular, animation requires the use of additional threads.

23.2 Drawing in a Graphics Window

Figure 23.4 shows the window that will be maintained by our second graphical program. Clicking the buttons will add and remove colored balls from the window. The balls will be stationary for now, but we'll add motion later.

An applet for this program is available on the web at:

```
http://www.cs.amherst.edu/lam/applets/Balls/balls.html
```

The Java files are available in the directory

```
http://www.cs.amherst.edu/lam/applets/Balls
```

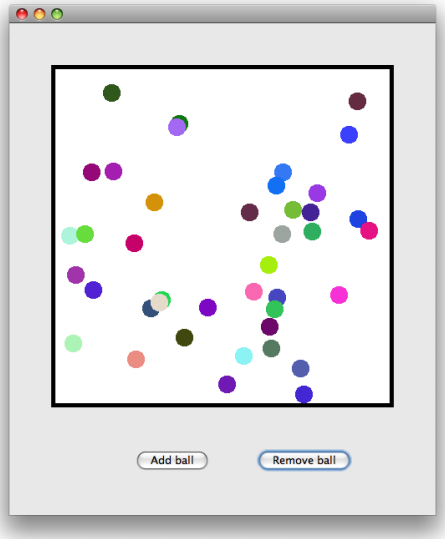

Figure 23.4: A graphical window containing colored balls.

Figure 23.5 shows the code for class `BallPanel`. A `BallPanel` object is associated with the entire contents of the window. Class `BallApp`, similar to `CounterApp` (Figure 23.3), creates a `BallPanel` object and calls `setContentPane` on it.

A `BallPanel` contains several subcomponents:

- A `BallCanvas`, the drawable area in which the balls appear. We'll discuss this subcomponent in detail.
- Two `JButton`s.
- A `ButtonPanel` that holds the two buttons. This subpanel is implemented via an inner class.

The `ButtonPanel` uses a `BoxLayout` in `Y_AXIS` mode to place the two buttons in a horizontal row. The main panel uses a `BoxLayout` in `X_AXIS` mode to place the `ButtonPanel` below the `BallCanvas`.

Graphics Idea 7 *In order to have a panel appear the way you want, it is sometimes useful to define one or more subpanels.*

Method `actionPerformed` runs when either button is clicked. This method calls the correct method in `BallCanvas`, either `addBall` or `removeBall`.

```
1   import javax.swing.*;
2   import java.awt.*;
3   import java.awt.event.*;
4   import javax.swing.border.*;
5
6   public class BallPanel extends JPanel implements ActionListener {
7
8       private BallCanvas canvas;
9       private JButton addButton, removeButton;
10
11      private class ButtonPanel extends JPanel {
12          ButtonPanel() {
13              setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
14              addButton = new JButton("Add ball");
15              addButton.setFocusable(false);
16              add(addButton);
17              add(Box.createHorizontalStrut(50));
18              removeButton = new JButton("Remove ball");
19              removeButton.setEnabled(false);
20              removeButton.setFocusable(false);
21              add(removeButton);
22              setBorder(new EmptyBorder(50, 0, 0, 0));
23          }
24      }
25
26      public BallPanel() {
27          setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
28          add(canvas = new BallCanvas(this));
29          add(new ButtonPanel());
30          addButton.addActionListener(this);
31          removeButton.addActionListener(this);
32          setBorder(new EmptyBorder(50,50,50,50));
33      }
34
35      public void actionPerformed (ActionEvent e) {
36          if (e.getSource().equals(addButton))
37              canvas.addBall();
38          else
39              canvas.removeBall();
40      }
41
42      public void enableRemove() {
43          removeButton.setEnabled(true);
44      }
45
46      public void disableRemove() {
47          removeButton.setEnabled(false);
48      }
49  }
```

Figure 23.5: BallPanel.java

On lines 15 and 20 in Figure 23.5, we specify that the buttons in the `ButtonPanel` should not be *focusable*. *Keyboard focus* is a tricky topic that we'll discuss later. A button is normally focusable, meaning it might become the default button, the one that is highlit and that can be activated when the user types the enter key. In the ball example, the enter key will not be associated with either button.

Figure 23.6 shows the code for the class `BallCanvas`, which is responsible for maintaining and displaying the area inside the black square. `BallCanvas` is a subclass of `JPanel`.

On lines 21-23, size parameters are given for the component.

Graphics Idea 8 *The size and appearance of a* `JPanel` *(and of other graphical components) is determined in a complex way. Every component can have preferred size. The layout manager (e.g.,* `BoxLayout`*) attempts to place all the graphical elements so that they have their preferred sizes. If the overall window is resized by the user, the layout manager will run again and will redo the layout. Maximum and/or minimum sizes can also be given, setting limits on the extent of resizing. Each size is expressed via using a* `Dimension` *object, which incorporates a width and a height.*

In the constructor for `BallCanvas`, all three sizes are set to the same value, meaning that no resizing is possible. The border is considered part of the component, so setting the sizes to be `SIZE + 2*BORDER_WIDTH` leaves a drawable area with width and height `SIZE`. The constructor also sets a background color for the component and sets a border.

(It's important to avoid overusing maximum and minimum sizes, because the layout manager needs flexibility in order to do its work. Sometimes it is simply impossible to honor all size requests, for example if a large window of fixed size is placed inside a smaller window of fixed size. In that case, the smaller window would let users see only part of the larger window.)

`BallCanvas` uses a linked list of `Ball` objects to record the configuration that should be displayed in the window. Methods `addBall` and `removeBall` are called whenever the appropriate buttons are clicked and `paintComponent` is called whenever the component needs to be redrawn.

Graphics Idea 9 *Methods that respond to events should call method* `repaint` *when they want to request that a component be redrawn on the screen.*

Methods `addBall` and `removeBall` both work by first changing the list of balls and then calling `repaint`. They also call methods `panel.enableRemove()` and

```
1   import javax.swing.*;
2   import java.awt.*;
3   import java.util.LinkedList;
4   import javax.swing.border.*;
5
6   public class BallCanvas extends JPanel {
7
8       final private static int BALL_DIAMETER = 20;
9       final private static int SIZE          = 400;
10      final private static int BORDER_WIDTH  = 5;
11
12      private BallPanel panel;
13      private LinkedList<Ball> balls = new LinkedList<Ball>();
14
15      public BallCanvas(BallPanel p) {
16
17          panel = p;
18          Ball.initialize(SIZE, BORDER_WIDTH);
19
20          int fullSize = SIZE + 2*BORDER_WIDTH;
21          setPreferredSize(new Dimension(fullSize, fullSize));
22          setMaximumSize(new Dimension(fullSize, fullSize));
23          setMinimumSize(new Dimension(fullSize, fullSize));
24
25          setBackground(Color.WHITE);
26          setBorder(new LineBorder(Color.BLACK, BORDER_WIDTH));
27      }
28
29      public void addBall() {
30          balls.add(new Ball(BALL_DIAMETER));
31          panel.enableRemove();
32          repaint();
33      }
34
35      public void removeBall() {
36          balls.removeLast();
37
38          if (balls.size() == 0)
39              panel.disableRemove();
40          repaint();
41      }
42
43      public void paintComponent(Graphics g) {
44          super.paintComponent(g);
45          for (Ball b : balls)
46              b.paint(g);
47      }
48  }
```

Figure 23.6: BallCanvas.java

`panel.disableRemove()` so that the panel will know when to enable or disable the remove button.

Graphics Idea 10 *Method* `paintComponent` *is responsible for drawing on the screen.*

A subclass of `JPanel` can supply a `paintComponent` method to draw lines, shapes, and text in the graphics window. No `paintComponent` method is needed if the entire scene consists of a background color plus some collection of subcomponents.

If present, method `paintComponent` must be `public` and `void`, and it must take one parameter of type `Graphics`. The `Graphics` object, often named `g`, is used in the method calls that do drawing. The first line in `paintComponent` should always be

```
super.paintComponent(g);
```

This calls method `paintComponent` in `JFrame`, which paints the background.

In Figure 23.6, this required call appears on line 44. It is followed by a loop that makes calls requesting that each `Ball` object draw itself in the window.

Figure 23.7 shows the code for class `Ball`. Before examining the details of this code, it is useful to understand the coordinate system used for drawing.

Graphics Idea 11 *In any component, pixel locations are given by a pair* (x, y)*, with location* $(0, 0)$ *being the upper left corner. The* x *coordinate increases as one moves to the right, and the* y *coordinate increases as one moves down.*

The maximum x coordinate is one less than the width of the window, and the maximum y coordinate is one less than the height. To draw a line from point $(x0, y0)$ to point $(x1, y1)$, you can simply write

```
g.drawLine(x0, y0, x1, y1);
```

When a `Ball` is constructed, a diameter is specified. A random position $(x0, y0)$ is generated for the ball, with coordinates chosen in the range $[0, (arenaSize - diameter - 1)]$. In Java graphics, all ovals (of which a circle is a special case) are considered be inscribed within an imaginary rectangle. The upper left corner of the imaginary rectangle is considered to be the location of the oval, even though that location isn't even contained in the oval. By setting $(arenaSize - diameter - 1)$ as the upper limit on the coordinates, we ensure that each ball falls inside the desired area.

```
1   import java.awt.*;
2
3   public class Ball {
4
5       private static int arenaSize;
6       private static int borderWidth;
7
8       private int x0, y0;
9       private Color color;
10      private int diameter;
11
12      public static void initialize(int a, int b) {
13          arenaSize = a;
14          borderWidth = b;
15      }
16
17      public Ball(int d) {
18          diameter = d;
19          x0 = (int)(Math.random() * (arenaSize - diameter));
20          y0 = (int)(Math.random() * (arenaSize - diameter));
21          color = new Color(rand255(), rand255(), rand255());
22      }
23
24      public void paint(Graphics g) {
25          g.setColor(color);
26          Point p = getPosition();
27
28          g.fillOval(p.x+borderWidth, p.y+borderWidth, diameter, diameter);
29      }
30
31      private Point getPosition() {
32          return new Point (x0, y0);
33      }
34
35      private int rand255() {
36          return (int)(Math.random()*256);
37      }
38  }
```

Figure 23.7: Ball.java

Each ball is assigned a random color that is created on line 21. The three arguments to the `Color` constructor are red, green, and blue intensities in the range $[0, 255]$. The code on line 19 uses random numbers for those intensities, and hence a random color is generated.

Method `paint` on line 24 is called from method `paintComponent` in `BallCanvas`. It sets the chosen color and draws the ball on the screen. (Note that the coordinates are adjusted by adding `borderWidth` to account for the fact that the border is drawn inside the component.) Class `Point`, used on line 26, is part of the graphics library.

23.3 The Relationship Between `repaint` and `paintComponent`

Suppose you have created a class for a graphical element, such as `BallCanvas`, that has a `paintComponent` method. Let's review what should happen in response to a button click or other event:

1. An event-handling method, such `actionPerformed`, begins running.
2. It, perhaps using other methods, updates data structures for the graphical component. (In the `Ball` example, the list of balls was updated.)
3. A call is made to `repaint`, requesting that the component be redrawn.
4. At some time in the near future, `paintComponent` runs and redisplays the component.

The distinction between repaint requests and actual painting is made because repaint requests can arise for many reasons. Suppose, for example, that one window is partially covering another and that the top window is closed. A repaint request would be generated automatically to repaint the part of the lower window that was exposed.

The possibility of a delay between a repaint request and the actual painting permits the Java graphics system to work more efficiently. If many events occur in quick succession, a single `paintComponent` call can be used to display all the updates.

Graphics Idea 12 *All methods that run in the event thread,* including `paintComponent`*, should do their work "quickly."*

Methods that run in the event thread should not do things that might cause indefinite delays. For example, they should not pause (via a sleep or wait operation), read from the keyboard, or try to make a connection to another computer.

It's tempting to try to achieve animation by using sleep calls or long loops within `paintComponent`. Don't do this! (We'll discuss alternatives later.) If the event thread becomes stuck in a method, no events can be processed and the graphics window will not be updated.

23.4 Using Mouse Events

Figure 23.8 shows the window that will be maintained by our next graphical program. The user can draw by pressing the mouse button mouse and dragging within the window. An applet for this program is available on the web at:

```
http://www.cs.amherst.edu/lam/applets/Scribble/scribble.html
```

The Java files are available in the directory

```
http://www.cs.amherst.edu/lam/applets/Scribble
```

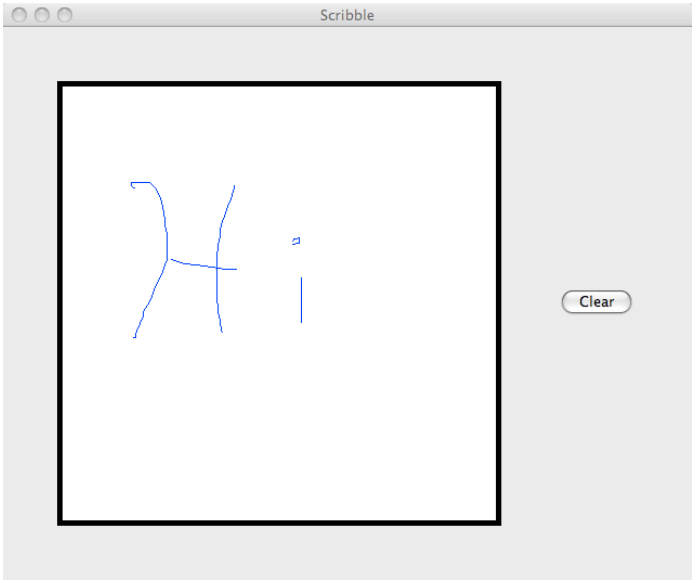

Figure 23.8: A graphical window supporting drawing with the mouse.

Code for class `ScribbleCanvas`, the class corresponding to the drawable area, is shown in Figure 23.9. The key data structure is `list`, a linked list of points. Whenever the mouse button is pressed, that point is added to the list. When the mouse is dragged from one point to another, the new point is added to the list.

```
1   import javax.swing.*;
2   import java.awt.*;
3   import java.awt.event.*;
4   import java.util.*;
5   import javax.swing.border.*;
6
7   public class ScribbleCanvas extends JPanel {
8
9       private LinkedList<Point> list = new LinkedList<Point>();
10
11      public ScribbleCanvas(int size) {
12          setPreferredSize(new Dimension(size,size));
13          setBackground(Color.WHITE);
14          addMouseMotionListener (new MyMotionListener());
15          addMouseListener (new MyMouseListener());
16          setBorder(new LineBorder(Color.BLACK, 5));
17      }
18
19      public void clear() {
20          list.clear();
21          repaint();
22      }
23
24      private void addPoint(Point p) {
25          list.add(p);
26      }
27
28      private class MyMouseListener extends MouseAdapter {
29          public void mousePressed(MouseEvent e) {
30              addPoint(e.getPoint());
31          }
32          public void mouseReleased(MouseEvent e) {
33              addPoint(null);
34          }
35      }
36
37      private class MyMotionListener extends MouseMotionAdapter {
38          public void mouseDragged (MouseEvent e) {
39              addPoint(e.getPoint());
40              repaint();
41          }
42      }
43
44      public void paintComponent( Graphics g ) {
45          super.paintComponent(g);
46
47          g.setColor(Color.blue);
48          Point prev = null;
49
50          for (Point p : list) {
51              if (prev != null && p != null)
52                  g.drawLine(prev.x, prev.y, p.x, p.y);
53              prev = p;
54          }
55      }
56  }
```

Figure 23.9: `ScribbleCanvas.java`

When the mouse is released, `null` is added to the list. The list is used in method `paintComponent` whenever the scene needs to be redrawn.

`ScribbleCanvas` handles mouse events via two listeners: a `MouseListener` and a `MouseMotionListener`.

Graphics Idea 13 *An easy way to implement a* listener *is to extend the corresponding* adapter*.*

`MyMouseListener` is a subclass of `MouseAdapter` and `MyMotionListener` is a subclass of `MouseMotionAdapter`. The problem with implementing a listener directly is that often a long list of methods is needed. A `MouseListener`, for example, requires five methods. An adapter implements a listener by providing trivial implementations of all of the methods. One can extend the adapter and provide a useful implementation for a small subset of the methods. In this case, we provide methods `mousePressed` and `mouseReleased` in `MyMouseListener` and method `mouseDragged` in `MyMotionListener`.

Code for class `ScribblePanel`, the class corresponding to main panel, is shown in Figure 23.10. This code is similar to that in `BallPanel` (Figure 23.5). The new code uses a `FlowLayout` instead of a `BoxLayout`. A `FlowLayout` arranges the components of a `JPanel` in one or more rows, based on the size of the components and the size of the overall panel. The parameters used in `ScribblePanel` specify that each row should be centered (from left to right) in the panel, and that there should be 50-pixel gaps between components both horizontally and vertically.

Graphics Idea 14 *A* `FlowLayout` *offers a quick way to format a* `JPanel` *without worrying about the details of component placement.*

If you choose, you can use a `FlowLayout` while developing a first version of a program and then move to a different layout later.

23.5 Animation

We now explore ways of animating a graphical program. Our first example is an animated version of the ball program from section 23.2.

An applet for this program is available on the web at:

```
http://www.cs.amherst.edu/lam/applets/Bounce/bounce.html
```

The Java files are available in

```
http://www.cs.amherst.edu/lam/applets/Bounce
```

```
1   import javax.swing.*;
2   import java.awt.*;
3   import java.awt.event.*;
4   import javax.swing.border.*;
5
6   public class ScribblePanel extends JPanel {
7
8       private ScribbleCanvas theCanvas;
9
10      public ScribblePanel (int size) {
11
12          super(new FlowLayout(FlowLayout.CENTER, 50, 50));
13
14          add (theCanvas = new ScribbleCanvas(size));
15
16          JButton clearButton = new JButton ("Clear");
17          clearButton.addActionListener (new MyActionListener());
18          clearButton.setFocusable(false);
19          add(clearButton);
20      }
21
22      private class MyActionListener implements ActionListener {
23          public void actionPerformed (ActionEvent e) {
24              theCanvas.clear();
25          }
26      }
27  }
```

Figure 23.10: `ScribblePanel.java`

The window is very similar to that shown in Figure 23.4 on page 195, but there are two additional buttons to start and stop the motion of the balls.

Figure 23.11 shows the code for the class `BounceCanvas`, which is an updated version of class `BallCanvas`, shown in Figure 23.6 on page 198.

To achieve animation, this code creates an `Animator` object on line 15 and uses it in various ways. On line 16, the `Animator` becomes the basis of a new thread. (We discuss details of creating and running threads shortly.) One of the tasks of the `Animator` object will be to keep track of the elapsed time for the animation, that is, how long the animation has been running. Any time during which the animation is stopped is not considered part of the elapsed time. Method `animator.startClock` is called whenever the Start button is clicked, and `animator.stopClock` is called whenever the Stop button is clicked. Method `animator.elapsedTime` returns the elapsed time in nanoseconds as a `long`, a 64-bit integer. The calls to create balls (line 39) and paint them (line 56) both rely on knowing the current elapsed time.

Figure 23.12 shows updated code for class `Ball`. The key idea is that an angle and velocity is assigned to each ball when it is created. To compute the location after some elapsed time, we first compute the location *as if there were no walls*. It is then easy to convert the location into one that includes the effect of the walls.

Figure 23.13 shows code for class `Animator`. An `Animator` object has two tasks: 1) keeping track of elapsed time, and 2) repeatedly invoking the `repaint` method on the `BounceCanvas` so that there is an illusion of motion.

The time-keeping code relies on calls to `System.nanoTime`, a method that returns the number of nanoseconds that have passed since some unspecified time t_0. (Time t_0 might even be in the future, in which case `nanoTime` returns a negative number.) While `nanoTime` is useless in answering the question "What time is it?", pairs of calls can be used to determine elapsed time, i.e., the time between the calls.

Using calls to `nanoTime`, we can implement `startClock`, `stopClock` and `elapsedTime` by using three variables:

- `running`: a boolean indicating whether the clock is running
- `clockTime1`: the time, as returned by `nanotime`, at which the clock last started
- `elapsedTime1`: the elapsed time when the clock last stopped

Based on these variables, it is straightforward to verify that methods `startClock`, `stopClock` and `elapsedTime` work in the desired manner.

```
1   public class BounceCanvas extends JPanel {
2
3       final private static int BALL_DIAMETER = 20;
4       final private static int SIZE          = 400;
5       final private static int MAX_VELOCITY  = 300;   // max velocity, in pixels per second
6       final private static int BORDER_WIDTH  = 5;
7
8       private BouncePanel parent;
9       private LinkedList<Ball> balls = new LinkedList<Ball>();
10      private Animator animator;
11
12      public BounceCanvas(BouncePanel f) {
13
14          parent = f;
15          animator = new Animator(this);
16          (new Thread(animator)).start();
17          Ball.initialize(SIZE, MAX_VELOCITY, BORDER_WIDTH);
18          int fullSize = SIZE + 2*BORDER_WIDTH;
19          setPreferredSize(new Dimension(fullSize, fullSize));
20          setMaximumSize(new Dimension(fullSize, fullSize));
21          setMinimumSize(new Dimension(fullSize, fullSize));
22          setBackground(Color.WHITE);
23          setBorder(new LineBorder(Color.BLACK, BORDER_WIDTH));
24      }
25
26      public void start() {
27          animator.startClock();
28      }
29
30      public void stop() {
31          animator.stopClock();
32      }
33
34      private long getTime() {
35          return animator.elapsedTime();
36      }
37
38      public void addBall() {
39          balls.add(new Ball(BALL_DIAMETER, getTime()));
40          parent.enableRemove();
41          repaint();
42      }
43
44      public void removeBall() {
45          balls.removeLast();
46
47          if (balls.size() == 0)
48              parent.disableRemove();
49          repaint();
50      }
51
52      public void paintComponent(Graphics g) {
53          super.paintComponent(g);
54          long t = animator.elapsedTime();
55          for (Ball b : balls)
56              b.paint(g, t);
57      }
58  }
```

Figure 23.11: BounceCanvas.java

```
1   class Ball {
2
3       private static int arenaSize;
4       private static double maxVelocity;
5       private static int borderWidth;
6       private int x0;
7       private int y0;
8       private long   t0;
9       private double vx;
10      private double vy;
11      private Color color;
12      private int diameter;
13
14      public static void initialize (int size, int maxv, int bw) {
15          arenaSize = size;
16          maxVelocity = maxv / 1000000000.0;
17          borderWidth = bw;
18      }
19
20      public Ball(int d, long t) {
21          diameter = d;
22          x0 = (int)(Math.random() * (arenaSize - diameter));
23          y0 = (int)(Math.random() * (arenaSize - diameter));
24          t0 = t;
25          color = new Color(rand255(), rand255(), rand255());
26          double angle = Math.random() * 2 * Math.PI;
27          double velocity = Math.random() * maxVelocity;  // pixels per nanosecond
28          vx = velocity * Math.cos(angle);
29          vy = velocity * Math.sin(angle);
30      }
31
32      private static int rand255() {
33          return (int)(Math.random()*256);
34      }
35
36      private Point getPosition (long t) {  /* location at time t */
37          return new Point (convert(x0 + vx*(t-t0)), convert(y0 + vy*(t-t0)));
38      }
39
40      private int convert (double x) {  /* converts a coordinate that ignores
41                                           the effect of the walls into one
42                                           that lies within the arena */
43
44          int effectiveSize = arenaSize - diameter;
45          int windows = (int) Math.floor(x / effectiveSize);
46          double result;
47
48          if (windows % 2 == 0)
49              result = x - (windows*effectiveSize);
50          else
51              result = effectiveSize*(1+windows) - x;
52          return (int) result;
53      }
54
55      public void paint (Graphics g, long t) {
56          Point p = getPosition(t);
57          g.setColor(color);
58          g.fillOval(p.x+borderWidth, p.y+borderWidth, diameter, diameter);
59      }
60  }
```

Figure 23.12: Ball.java

```
1   import java.awt.*;
2
3   public class Animator implements Runnable {
4
5       private static final int DELAY = 30;  // time between redraws in milliseconds
6
7       private long clockTime1;   // see method elapsedTime()
8       private long elapsedTime1;
9       private boolean running;
10
11      private Component component;
12
13      public Animator (Component component) {
14          this.component = component;
15      }
16
17      public synchronized void run() {
18
19          try {
20              while (true) {
21
22                  if (running) {
23                      wait(DELAY);
24                  }
25                  else {
26                      wait();
27                  }
28                  component.repaint();
29              }
30          }
31          catch (InterruptedException e) {
32              // run() must catch these
33          }
34      }
35
36      public synchronized void startClock() {
37          clockTime1 = System.nanoTime();
38          running = true;
39          notify();
40      }
41
42      public synchronized void stopClock() {
43          elapsedTime1 = elapsedTime();
44          running = false;
45      }
46
47      public synchronized long elapsedTime() {
48          if (!running) return elapsedTime1;
49          else return elapsedTime1 + System.nanoTime() - clockTime1;
50      }
51  }
```

Figure 23.13: Animator.java

Class `Animator` implements interface `Runnable` by providing a method with the following header:

```
public void run()
```

Recall that `BounceCanvas` (Figure 23.11) contained the following lines:

```
animator = new Animator(this);
(new Thread(animator)).start();
```

A new thread of execution (represented by a `Thread`) can be created based on any `Runnable` object. If method `start` is applied to the `Thread`, the `run` method begins running in the new thread, and the old thread (the one that called `start`) continues its work, falling through to the whatever statement follows the call to `start`.

The heart of method `run` in `Animator` is an infinite `while` loop. If `running` is true, the thread waits (sleeps) for 30 milliseconds and then does a `repaint`. The continuous stream of `repaint` requests ensures that the window is redrawn frequently enough to give the illusion of motion.

Method `wait` is implemented by the Java system in a very efficient manner. When a thread is in `wait`, it uses no processor cycles, i.e. the whole processor can be used for other threads or other processes. So effectively the animation thread: 1) sleeps for 30 milliseconds (a long time by computer standards), 2) wakes up and quickly issues a `repaint` request, and 3) goes back to sleep. The existence of this thread has almost no impact on the system resources available to other threads and processes.

If `running` is false, the animation thread calls `wait()` on line 26. The no-arguments version of `wait` enters a permanent wait. This raises an obvious question, "If it's permanent, how does the thread ever wake up?" The secret is the call to `notify` in method `startClock`. When the start button is clicked, the event thread runs `startClock`. The call to `notify` causes the animation thread to leave its `wait`.

23.6 Synchronization

In class `Animator` (Figure 23.13), the methods `run`, `startClock`, `stopClock` and `elapsedTime` are labelled as `synchronized`. If one thread (such as the event thread) is using one of these methods, no other thread (such as the animation thread) can use any of them. This implies that the shared variables can only be used by one thread at a time.

The process of synchronization can be described more precisely in the following way. In Java, every object is a possible *locus of synchronization*. In effect, every object has a *lock bit* and a way of keeping track of which thread (if any) *holds*

the lock. When a thread calls a nonstatic `synchronized` method on some object, it tries to acquire the lock for the object. If one thread holds the lock, no other thread can enter a method synchronized on the same object. In the animated ball program, there is a single `Animator` object, and its lock bit is used as the basis of the synchronization in Figure 23.13.

In addition to writing methods labelled with the keyword `synchronized`, synchronization can be achieved using a *synchronized statement*. Here's an example:

```
synchronized (o) { ... }
```

Object `o` is the locus of synchronization controlling access to the block of code within the braces. If one thread is within the block, no other thread can enter a method or block synchronized on `o`.

The two `wait` methods are declared as nonstatic methods in class `Object`, which means that they can be called in any nonstatic method or in conjunction with any object. Any thread that calls `wait` must hold the lock for the object, i.e., it must be in a method or block synchronized on that object. The typical use of `wait` is a simple call to `wait()` or `wait(DELAY)` within a synchronized method.

Method `notify` is similar. It's a nonstatic method in `Object`. Any thread that calls it must hold the lock for the object, and the typical use is a simple call within a synchronized method. The effect of `notify` is to wake up some thread (if there is one) that is waiting on the same object.

Here's the interesting twist. When `wait` is called, the lock is freed so that some other thread can enter a synchronized method that depends on the same lock. This is essential, because otherwise no thread could do a `notify`. The waiting thread must reacquire the lock before leaving the wait method.

Suppose the bouncing ball program is not running, i.e., the start button has not been clicked. The animation thread (the one running `run`) becomes stuck at the call to `wait()` on line 26 and gives up the lock. Eventually the user clicks the start button. The event thread acquires the lock, runs `startClock`, records the time, sets `running` to `true` and calls `notify`. The call to `notify` wakes the animation thread. The event thread leaves method `startClock` and frees the lock. The animation thread can then reacquire the lock and move out of the wait.

A common use of synchronization is to protect access to a shared data structure. Suppose, for example, that an animation thread was allowed to add or remove balls, via calls to `addBall` or `removeBall`. That would open the possibility that the event and animation threads might try to modify the list of balls simultaneously. Similarly, the animation thread might try to change the list while the event thread tried to use the list in `paintComponent`. To prevent this, methods `addBall`, `removeBall`, and `paintComponent` should all be marked `synchronized`. More generally, whenever

there is a chance that two threads might use a data structure in inconsistent ways, synchronization should be used.

Appendix A

Common Errors

1. *Failing to save files after editing them.*

   Whenever you edit or make changes in a `.java` file, don't forget to save the modified file. Otherwise, the changes won't be used when you recompile the file, leading to confusion. Be sure you are saving in the right directory.

2. *Failing to recompile after changing a* `.java` *file.*

   Until you recompile, any changes will have no effect.

3. *Omitting keywords or punctuation.* Be sure that parentheses, braces, and quotation marks are matched correctly.

4. *Misspelling keywords or identifiers.* Remember that Java distinguishes between upper and lower-case, so a variable `i` is distinct from a variable `I`. The keyword `class` can not be written `Class`.

5. *Failure to declare variables.* Each variable must be declared.

6. *Failure to initialize variables.* Every variable must contain a value before it is used.

7. *Errors with comments.* Beware of problems caused by comments, especially those delimited by `/*` and `*/`. Suppose you write

```
    /* this is a comment that I forgot to close
    i = 3;
}

/* here's a later comment */
```

The absence of the `*/` at the end of the first comment means that both the assignment and the right brace are effectively taken out of the program. The comment finally ends at the end of the second comment. The program is effectively missing a right-brace, which will cause an error later.

Appendix B

Compiler Error Messages

These are typical error messages generated by the Java compiler in Sun's Java Development Kit version 1.1.5. Similar messages are used on other systems and in more recent versions of the Sun/Oracle SDK.

1. `Can't read`

```
> javac TempCalc.java
error: Can't read: TempCalc.java
```

Probably you didn't save the `.java` file or saved it in the in the wrong directory.

2. `Can't find class`

```
>javac TempCalc.java
Can't find class TempCalc
```

Probably you didn't compile the `.java` file or are in the wrong directory.

3. `Undefined variable or class name`

```
./TempCalc.java:13: Undefined variable or class name: Keyboard
    fahrenheit = Keyboard.nextDouble();     // reads from the keyboard
                 ^
```

Probably you mistyped a name. In this case, the class is intended to be `keyboard`, not `Keyboard`.

4. `Keyword expected`

```
./TempCalc.java:1: 'class' or 'interface' keyword expected.
public TempCalc {
       ^
```

   A keyword is missing or is perhaps misspelled or miscapitalized.

5. `Public class must be defined in a file called...`

```
./TempCalc.java:1: Public class tempCalc must be defined in a
                                file called "tempCalc.java".
public class tempCalc {
             ^
```

   The name of a public class must match the filename. Capitalization is important.

6. `Invalid method declaration; return type required.`

```
./TempCalc.java:5: Invalid method declaration; return type required.
  public static main (String[] args) {
                ^
```

   Every method must indicate what type of value it returns; for `main` the type should be `void`.

7. `{ expected`

```
./TempCalc.java:5: '{' expected.
  public static void main (String[] args)
                                          ^
```

   The `{` that is part of the method declaration is missing. In general, this kind of message means that expected item is missing. If you can't find the error, check the previous line and the next one.

8. `Incompatible type`

```
./TempCalc.java:12: Incompatible type for =. Can't convert double to
                                              java.lang.Double.
    fahrenheit = keyboard.nextDouble();     // reads from the keyboard
               ^
```

This error means that you're trying to assign a value of some type to a variable that can't hold that type. In this case the problem was that `fahrenheit` was declared to have type `Double`, a misspelling of `double`. Check your declarations when you get this kind of error.

Here's another example:

```
./TempCalc.java:14: Incompatible type for -. Can't convert
                    java.lang.Double to int.
    celsius = (fahrenheit-32) * (5.0/9.0);
                          ^
```

In this case, `fahrenheit` couldn't be converted to an `int` for subtraction.

9. `Undefined ... name`

```
./TempCalc.java:11: Undefined variable, class, or package
                    name: system
  system.out.print("Enter a temperature in Fahrenheit degrees: ");
  ^
```

The name `system` is unknown. The correct name is `System`.

10. `Method not found`

```
./TempCalc.java:11: Method Print(java.lang.String) not found in
                    class java.io.PrintStream.
  System.out.Print ("Enter a temperature in Fahrenheit degrees: ");
                   ^
```

Again, this is a misspelling. The correct method is `print`.

11. `String not terminated at end of line`

```
./TempCalc.java:11: String not terminated at end of line.
  System.out.print ("Enter a temperature in Fahrenheit degrees: );
                     ^
```

Each quoted string must be contained in a single line; in this case the closing quote mark was mistakenly omitted.

12. `Invalid character constant`

```
./TempCalc.java:11: Invalid character constant.
  System.out.print('Enter a temperature in Fahrenheit degrees: ');
                   ^
```

Strings must use double quotes. Single quote marks are used for characters, a different type.

13. `Invalid type expression`

```
./TempCalc.java:11: Invalid type expression.
  System.out.print ("Enter a temperature in Fahrenheit degrees: ")
                  ^
```

This message usually means that something is missing, such as a semicolon or parenthesis. Often the problem is on the previous line or the next one.

14. `Variable is already defined in this method`

```
./TempCalc.java:12: Variable 'fahrenheit' is already defined
                    in this method.
    int fahrenheit;
        ^
```

The variable was declared earlier in the same method.

15. `Undefined variable`

```
./TempCalc.java:15: Undefined variable: kelvin
    kelvin = celsius + 273.15;
    ^
```

A variable was never declared.

16. `Attempt to reference method as instance variable`

```
./TempCalc.java:12: Attempt to reference method nextDouble
                    in class Scanner as an instance variable.
    fahrenheit = keyboard.nextDouble;     // reads from the keyboard
                  ^
```

The parentheses were omitted in the call to `nextDouble`.

17. `Variable may not have been initialized`

```
./TempCalc.java:14: Variable fahrenheit may not have been
                    initialized.
    celsius = (fahrenheit-32) * (5.0/9.0);
               ^
```

No value was previously assigned to `fahrenheit`. It is therefore illegal to use it in an expression.

18. `Invalid left-hand side`

```
./TempCalc.java:14: Invalid left hand side of assignment.
    celsius = (fahrenheit-32) * (5.0 / 9.0)
                               ^
```

Something is wrong with the left-hand side of an assigment. In this case, the problem is actually a missing semicolon. The assignment on the *next line* is therefore disrupted, leading to a confusing message.

19. `Class not found`

```
./TempCalc.java:20: Class string not found in type declaration.
    string s = keyboard.nextLine();
        ^
```

The class name `String` is misspelled.

Appendix C

Frequently Asked Questions about Objects

1. *When do you want to have an object?*

   First and foremost, use objects to bundle together related variables. For example,

   - A BankAccount object can hold an owner and a balance.
   - A Student object can hold a name, a box number, and a year.
   - A Cootie object can hold a number of legs, eyes, etc.
   - A Flight object can hold an origin, a destination, and a distance

   If you have a group of related variables, *define a new kind of object*, that is, *write a new class*.

2. *What's an instance variable?*

   An instance variable is declared within a class and outside of the methods. It must be non-static (i.e. not labeled as static). You should create one instance variable for each attribute (e.g. name, box, year) that is an intrinsic part of the object.

3. *What's a constructor?*

   A constructor is a special method with a header similar to the following:

   ```
   public Airport (String name) { ...
   ```

   The name must match the name of the class. When you write

   ```
   Airport boston = new Airport ("Boston");
   ```

space is allocated for a new object, the constructor is executed, and variable `boston` is set to point to the new object.

Most constructors simply copy values into instance variables, but some allocate arrays and do other initialization. The constructor should do whatever is necessary to set the instance variables to sensible values.

4. *Can you have two constructors in the same class?*

   Yes! This is fine as long as the parameters lists have different numbers or types of parameters. Whichever one matches the argument list given in the `new` is used.

5. *Can you have no constructors in a class?*

   Yes. When you write "`new A()`", an object is allocated and the instance variables are given default values, 0 for ints or doubles, `false` for booleans, or `null` for objects.

6. *Why create a non-static method?*

   You can create a non-static method if there is some operation that you would like to do to an object. For example, the String class has non-static methods `length()` and `toLowerCase()`, the Cootie class has non-static method `takeTurn()`, the Airport class has method `addFlight()`, and so on. They would be called by writing

   ```
   int i = s.length();
   String t = s.toLowerCase();
   c.takeTurn();
   a.addFlight(f);
   ```

   assuming that `a`, `s`, `c`, and `f` had the right types.

7. *Why not use static methods for the same purpose?*

   You can, but it makes things a bit more confusing. The calls would become

   ```
   int i = String.length(s);
   String t = String.toLowerCase(s);
   Cootie.takeTurn(c);
   Airport.addFlight(a,f);
   ```

   The longer name of each method would give the name of the class containing the method, and the object itself would have to be passed as a parameter.

   The static methods would themselves be more confusing. Recall that the code for our non-static method addFlight is:

```
public void addFlight (Flight f) {

    if (flightCount == flights.length)
        throw new RuntimeException ("Airport " + name
            + " has hit its flight limit");

    flights[flightCount++] = f;
 }
```

We can refer to `name`, `flights` and `flightCount` because they are associated with the Airport object to which the method is being applied. If you really wanted to use a static method, it would be trickier:

```
public static void addFlight (Airport a, Flight f) {

    if (a.flightCount == a.flights.length)
        throw new RuntimeException ("Airport " + a.name
            + " has hit its flight limit");

    a.flights[a.flightCount++] = f;
}
```

Non-static methods simply make things easier, and you should use them when you can.

8. *Why should we have any static methods at all?*

   Sometimes the method isn't associated with a particular object. Examples:

   - A main method.
   - A method, such as `sqrt()`, or `cos()`, in the Math class.
   - Methods that are in the main class and are called by the main method.

9. *What does "private" mean?*

   A method or variable that is private can only be used by methods within the same class.

10. *What does "public" mean?*

    A method or variable that is public can be used by methods in any class.

11. *What's a class variable?*

    A class variable is a static variable declared within a class and outside of methods. They are variables that associated with the class as a whole and not with particular objects. The most important use of such variables is to share information among static methods without having to pass as many parameters and return values. You should use class variables in moderation, because they do hide interactions between different methods.

12. *When do you want to say* `a.f(...)`?

    Do this to apply method `f` to object `a`. Suppose `a` is declared in class `A`; there must be a method `f` (with parameters of the right types) in class `A`. If `f` is private, this call can only be made from within the same class.

13. *When do you want to say* `A.f(...)`?

    Do this to call a static method called `f` in class `A`. If `f` is private, this call can only be made from within `A`.

14. *When do you want to say* `f(...)`?

    Do this to call a static method called `f` in the same class. Also, if you're in a non-static method, you can use it to call another non-static method on the same object.

15. *When do you want to refer to a variable by writing* `a.i`*?*

    Do this to refer to an instance variable `i` that is within the object referenced by `a`. If `a` is private, this can only be used within the same class.

16. *When do you want to refer to a variable by writing* `A.i`*?*

    This is very unusual and would refer to a public class variable in another class.

17. *When do you want to refer to a variable by writing* `i`*?*

    This occurs in one of the following situations:

    - `i` is a local variable or a parameter.
    - You are in a non-static method and `i` is an instance variable for the current object or a class variable for the class.
    - You are in a static method and `i` is a class variable for the class.

18. *When do you write* this.i*?*

    Do this in a constructor if (1) there is a parameter or local called `i`, and (2) you want to refer to an instance variable of the same name.

Index

The index entries are organized into nine broad categories: API classes; concepts; constants and variables; examples; keywords; methods; operators; statements; and types.