

Chapter 4

Programming in the Large I: Subroutines

ONE WAY TO BREAK UP A COMPLEX PROGRAM into manageable pieces is to use *subroutines*. A subroutine consists of the instructions for carrying out a certain task, grouped together and given a name. Elsewhere in the program, that name can be used as a stand-in for the whole set of instructions. As a computer executes a program, whenever it encounters a subroutine name, it executes all the instructions necessary to carry out the task associated with that subroutine.

Subroutines can be used over and over, at different places in the program. A subroutine can even be used inside another subroutine. This allows you to write simple subroutines and then use them to help write more complex subroutines, which can then be used in turn in other subroutines. In this way, very complex programs can be built up step-by-step, where each step in the construction is reasonably simple.

As mentioned in Section 3.8, subroutines in Java can be either static or non-static. This chapter covers static subroutines only. Non-static subroutines, which are used in true object-oriented programming, will be covered in the next chapter.

4.1 Black Boxes

A SUBROUTINE CONSISTS OF INSTRUCTIONS for performing some task, chunked together and given a name. “Chunking” allows you to deal with a potentially very complicated task as a single concept. Instead of worrying about the many, many steps that the computer might have to go through to perform that task, you just need to remember the name of the subroutine. Whenever you want your program to perform the task, you just call the subroutine. Subroutines are a major tool for dealing with complexity.

A subroutine is sometimes said to be a “black box” because you can’t see what’s “inside” it (or, to be more precise, you usually don’t **want** to see inside it, because then you would have to deal with all the complexity that the subroutine is meant to hide). Of course, a black box that has no way of interacting with the rest of the world would be pretty useless. A black box needs some kind of *interface* with the rest of the world, which allows some interaction between what’s inside the box and what’s outside. A physical black box might have buttons on the outside that you can push, dials that you can set, and slots that can be used for passing information back and forth. Since we are trying to hide complexity, not create it, we have the first rule of black boxes:

The interface of a black box should be fairly straightforward, well-defined, and easy to understand.

Are there any examples of black boxes in the real world? Yes; in fact, you are surrounded by them. Your television, your car, your VCR, your refrigerator. . . . You can turn your television on and off, change channels, and set the volume by using elements of the television's interface—dials, remote control, don't forget to plug in the power—without understanding anything about how the thing actually works. The same goes for a VCR, although if the stories are true about how hard people find it to set the time on a VCR, then maybe the VCR violates the simple interface rule.

Now, a black box does have an inside—the code in a subroutine that actually performs the task, all the electronics inside your television set. The inside of a black box is called its *implementation*. The second rule of black boxes is that:

To use a black box, you shouldn't need to know anything about its implementation; all you need to know is its interface.

In fact, it should be possible to **change** the implementation, as long as the behavior of the box, as seen from the outside, remains unchanged. For example, when the insides of TV sets went from using vacuum tubes to using transistors, the users of the sets didn't even need to know about it—or even know what it means. Similarly, it should be possible to rewrite the inside of a subroutine, to use more efficient code, for example, without affecting the programs that use that subroutine.

Of course, to have a black box, someone must have designed and built the implementation in the first place. The black box idea works to the advantage of the implementor as well as of the user of the black box. After all, the black box might be used in an unlimited number of different situations. The implementor of the black box doesn't need to know about any of that. The implementor just needs to make sure that the box performs its assigned task and interfaces correctly with the rest of the world. This is the third rule of black boxes:

The implementor of a black box should not need to know anything about the larger systems in which the box will be used.

In a way, a black box divides the world into two parts: the inside (implementation) and the outside. The interface is at the boundary, connecting those two parts.

* * *

By the way, you should **not** think of an interface as just the physical connection between the box and the rest of the world. The interface also includes a *specification* of what the box does and how it can be controlled by using the elements of the physical interface. It's not enough to say that a TV set has a power switch; you need to specify that the power switch is used to turn the TV on and off!

To put this in computer science terms, the interface of a subroutine has a semantic as well as a syntactic component. The syntactic part of the interface tells you just what you have to type in order to call the subroutine. The semantic component specifies exactly what task the subroutine will accomplish. To write a legal program, you need to know the syntactic specification of the subroutine. To understand the purpose of the subroutine and to use it effectively, you need to know the subroutine's semantic specification. I will refer to both parts of the interface—syntactic and semantic—collectively as the *contract* of the subroutine.

The contract of a subroutine says, essentially, “Here is what you have to do to use me, and here is what I will do for you, guaranteed.” When you write a subroutine, the comments that you write for the subroutine should make the contract very clear. (I should admit that in practice, subroutines’ contracts are often inadequately specified, much to the regret and annoyance of the programmers who have to use them.)

For the rest of this chapter, I turn from general ideas about black boxes and subroutines in general to the specifics of writing and using subroutines in Java. But keep the general ideas and principles in mind. They are the reasons that subroutines exist in the first place, and they are your guidelines for using them. This should be especially clear in Section 4.6, where I will discuss subroutines as a tool in program development.

* * *

You should keep in mind that subroutines are not the only example of black boxes in programming. For example, a class is also a black box. We’ll see that a class can have a “public” part, representing its interface, and a “private” part that is entirely inside its hidden implementation. All the principles of black boxes apply to classes as well as to subroutines.

4.2 Static Subroutines and Static Variables

EVERY SUBROUTINE IN JAVA must be defined inside some class. This makes Java rather unusual among programming languages, since most languages allow free-floating, independent subroutines. One purpose of a class is to group together related subroutines and variables. Perhaps the designers of Java felt that everything must be related to something. As a less philosophical motivation, Java’s designers wanted to place firm controls on the ways things are named, since a Java program potentially has access to a huge number of subroutines created by many different programmers. The fact that those subroutines are grouped into named classes (and classes are grouped into named “packages”) helps control the confusion that might result from so many different names.

A subroutine that is a member of a class is often called a *method*, and “method” is the term that most people prefer for subroutines in Java. I will start using the term “method” occasionally; however, I will continue to prefer the more general term “subroutine” for static subroutines. I will use the term “method” most often to refer to non-static subroutines, which belong to objects rather than to classes. This chapter will deal with static subroutines almost exclusively. We’ll turn to non-static methods and object-oriented programming in the next chapter.

4.2.1 Subroutine Definitions

A subroutine definition in Java takes the form:

```

    <modifiers> <return-type> <subroutine-name> ( <parameter-list> ) {
        <statements>
    }

```

It will take us a while—most of the chapter—to get through what all this means in detail. Of course, you’ve already seen examples of subroutines in previous chapters, such as the `main()` routine of a program and the `paint()` routine of an applet. So you are familiar with the general format.

The *<statements>* between the braces, { and }, in a subroutine definition make up the *body* of the subroutine. These statements are the inside, or implementation part, of the “black box”,

as discussed in the previous section. They are the instructions that the computer executes when the method is called. Subroutines can contain any of the statements discussed in Chapter 2 and Chapter 3.

The *modifiers* that can occur at the beginning of a subroutine definition are words that set certain characteristics of the subroutine, such as whether it is static or not. The modifiers that you've seen so far are “**static**” and “**public**”. There are only about a half-dozen possible modifiers altogether.

If the subroutine is a function, whose job is to compute some value, then the *return-type* is used to specify the type of value that is returned by the function. We'll be looking at functions and return types in some detail in Section 4.4. If the subroutine is not a function, then the *return-type* is replaced by the special value **void**, which indicates that no value is returned. The term “void” is meant to indicate that the return value is empty or non-existent.

Finally, we come to the *parameter-list* of the method. Parameters are part of the interface of a subroutine. They represent information that is passed into the subroutine from outside, to be used by the subroutine's internal computations. For a concrete example, imagine a class named *Television* that includes a method named `changeChannel()`. The immediate question is: What channel should it change to? A parameter can be used to answer this question. Since the channel number is an integer, the type of the parameter would be **int**, and the declaration of the `changeChannel()` method might look like

```
public void changeChannel(int channelNum) { ... }
```

This declaration specifies that `changeChannel()` has a parameter named `channelNum` of type **int**. However, `channelNum` does not yet have any particular value. A value for `channelNum` is provided when the subroutine is called; for example: `changeChannel(17)`;

The parameter list in a subroutine can be empty, or it can consist of one or more parameter declarations of the form *type* *parameter-name*. If there are several declarations, they are separated by commas. Note that each declaration can name only one parameter. For example, if you want two parameters of type **double**, you have to say “double x, double y”, rather than “double x, y”.

Parameters are covered in more detail in the next section.

Here are a few examples of subroutine definitions, leaving out the statements that define what the subroutines do:

```
public static void playGame() {
    // "public" and "static" are modifiers; "void" is the
    // return-type; "playGame" is the subroutine-name;
    // the parameter-list is empty.
    . . . // Statements that define what playGame does go here.
}

int getNextN(int N) {
    // There are no modifiers; "int" in the return-type
    // "getNextN" is the subroutine-name; the parameter-list
    // includes one parameter whose name is "N" and whose
    // type is "int".
    . . . // Statements that define what getNextN does go here.
}

static boolean lessThan(double x, double y) {
    // "static" is a modifier; "boolean" is the
    // return-type; "lessThan" is the subroutine-name; the
```

```

    // parameter-list includes two parameters whose names are
    // "x" and "y", and the type of each of these parameters
    // is "double".
    . . . // Statements that define what lessThan does go here.
}

```

In the second example given here, `getNextN` is a non-static method, since its definition does not include the modifier “`static`”—and so it’s not an example that we should be looking at in this chapter! The other modifier shown in the examples is “`public`”. This modifier indicates that the method can be called from anywhere in a program, even from outside the class where the method is defined. There is another modifier, “`private`”, which indicates that the method can be called **only** from inside the same class. The modifiers `public` and `private` are called *access specifiers*. If no access specifier is given for a method, then by default, that method can be called from anywhere in the “package” that contains the class, but not from outside that package. (Packages were introduced in Subsection 2.6.4, and you’ll learn more about them later in this chapter, in Section 4.5.) There is one other access modifier, `protected`, which will only become relevant when we turn to object-oriented programming in Chapter 5.

Note, by the way, that the `main()` routine of a program follows the usual syntax rules for a subroutine. In

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

the modifiers are `public` and `static`, the return type is `void`, the subroutine name is `main`, and the parameter list is “`String[] args`”. The only question might be about “`String[]`”, which has to be a type if it is to match the syntax of a parameter list. In fact, `String[]` represents a so-called “array type”, so the syntax is valid. We will cover arrays in Chapter 7. (The parameter, `args`, represents information provided to the program when the `main()` routine is called by the system. In case you know the term, the information consists of any “command-line arguments” specified in the command that the user typed to run the program.)

You’ve already had some experience with filling in the implementation of a subroutine. In this chapter, you’ll learn all about writing your own complete subroutine definitions, including the interface part.

4.2.2 Calling Subroutines

When you define a subroutine, all you are doing is telling the computer that the subroutine exists and what it does. The subroutine doesn’t actually get executed until it is called. (This is true even for the `main()` routine in a class—even though **you** don’t call it, it is called by the system when the system runs your program.) For example, the `playGame()` method given as an example above could be called using the following subroutine call statement:

```
playGame();
```

This statement could occur anywhere in the same class that includes the definition of `playGame()`, whether in a `main()` method or in some other subroutine. Since `playGame()` is a `public` method, it can also be called from other classes, but in that case, you have to tell the computer which class it comes from. Since `playGame()` is a `static` method, its full name includes the name of the class in which it is defined. Let’s say, for example, that `playGame()` is defined in a class named `Poker`. Then to call `playGame()` from **outside** the `Poker` class, you would have to say

```
Poker.playGame();
```

The use of the class name here tells the computer which class to look in to find the method. It also lets you distinguish between `Poker.playGame()` and other potential `playGame()` methods defined in other classes, such as `Roulette.playGame()` or `Blackjack.playGame()`.

More generally, a *subroutine call statement* for a static subroutine takes the form

```
<subroutine-name>(<parameters>);
```

if the subroutine that is being called is in the same class, or

```
<class-name>.<subroutine-name>(<parameters>);
```

if the subroutine is defined elsewhere, in a different class. (Non-static methods belong to objects rather than classes, and they are called using object names instead of class names. More on that later.) Note that the parameter list can be empty, as in the `playGame()` example, but the parentheses must be there even if there is nothing between them.

4.2.3 Subroutines in Programs

It's time to give an example of what a complete program looks like, when it includes other subroutines in addition to the `main()` routine. Let's write a program that plays a guessing game with the user. The computer will choose a random number between 1 and 100, and the user will try to guess it. The computer tells the user whether the guess is high or low or correct. If the user gets the number after six guesses or fewer, the user wins the game. After each game, the user has the option of continuing with another game.

Since playing one game can be thought of as a single, coherent task, it makes sense to write a subroutine that will play one guessing game with the user. The `main()` routine will use a loop to call the `playGame()` subroutine over and over, as many times as the user wants to play. We approach the problem of designing the `playGame()` subroutine the same way we write a `main()` routine: Start with an outline of the algorithm and apply stepwise refinement. Here is a short pseudocode algorithm for a guessing game program:

```
Pick a random number
while the game is not over:
    Get the user's guess
    Tell the user whether the guess is high, low, or correct.
```

The test for whether the game is over is complicated, since the game ends if either the user makes a correct guess or the number of guesses is six. As in many cases, the easiest thing to do is to use a “`while (true)`” loop and use `break` to end the loop whenever we find a reason to do so. Also, if we are going to end the game after six guesses, we'll have to keep track of the number of guesses that the user has made. Filling out the algorithm gives:

```
Let computersNumber be a random number between 1 and 100
Let guessCount = 0
while (true):
    Get the user's guess
    Count the guess by adding 1 to guess count
    if the user's guess equals computersNumber:
        Tell the user he won
        break out of the loop
    if the number of guesses is 6:
        Tell the user he lost
        break out of the loop
    if the user's guess is less than computersNumber:
```

```

        Tell the user the guess was low
    else if the user's guess is higher than computersNumber:
        Tell the user the guess was high

```

With variable declarations added and translated into Java, this becomes the definition of the `playGame()` routine. A random integer between 1 and 100 can be computed as `(int)(100 * Math.random()) + 1`. I've cleaned up the interaction with the user to make it flow better.

```

static void playGame() {
    int computersNumber; // A random number picked by the computer.
    int usersGuess;      // A number entered by user as a guess.
    int guessCount;      // Number of guesses the user has made.
    computersNumber = (int)(100 * Math.random()) + 1;
        // The value assigned to computersNumber is a randomly
        // chosen integer between 1 and 100, inclusive.
    guessCount = 0;
    TextIO.putln();
    TextIO.put("What is your first guess? ");
    while (true) {
        usersGuess = TextIO.getInt(); // Get the user's guess.
        guessCount++;
        if (usersGuess == computersNumber) {
            TextIO.putln("You got it in " + guessCount
                + " guesses! My number was " + computersNumber);
            break; // The game is over; the user has won.
        }
        if (guessCount == 6) {
            TextIO.putln("You didn't get the number in 6 guesses.");
            TextIO.putln("You lose. My number was " + computersNumber);
            break; // The game is over; the user has lost.
        }
        // If we get to this point, the game continues.
        // Tell the user if the guess was too high or too low.
        if (usersGuess < computersNumber)
            TextIO.put("That's too low. Try again: ");
        else if (usersGuess > computersNumber)
            TextIO.put("That's too high. Try again: ");
    }
    TextIO.putln();
} // end of playGame()

```

Now, where exactly should you put this? It should be part of the same class as the `main()` routine, but **not** inside the `main` routine. It is not legal to have one subroutine physically nested inside another. The `main()` routine will **call** `playGame()`, but not contain it physically. You can put the definition of `playGame()` either before or after the `main()` routine. Java is not very picky about having the members of a class in any particular order.

It's pretty easy to write the `main` routine. You've done things like this before. Here's what the complete program looks like (except that a serious program needs more comments than I've included here).

```

public class GuessingGame {

    public static void main(String[] args) {
        TextIO.putln("Let's play a game. I'll pick a number between");
        TextIO.putln("1 and 100, and you try to guess it.");
    }
}

```

```

    boolean playAgain;
    do {
        playGame(); // call subroutine to play one game
        TextIO.put("Would you like to play again? ");
        playAgain = TextIO.getlnBoolean();
    } while (playAgain);
    TextIO.putln("Thanks for playing. Goodbye.");
} // end of main()

static void playGame() {
    int computersNumber; // A random number picked by the computer.
    int usersGuess;      // A number entered by user as a guess.
    int guessCount;      // Number of guesses the user has made.
    computersNumber = (int)(100 * Math.random()) + 1;
        // The value assigned to computersNumber is a randomly
        // chosen integer between 1 and 100, inclusive.
    guessCount = 0;
    TextIO.putln();
    TextIO.put("What is your first guess? ");
    while (true) {
        usersGuess = TextIO.getInt(); // Get the user's guess.
        guessCount++;
        if (usersGuess == computersNumber) {
            TextIO.putln("You got it in " + guessCount
                + " guesses! My number was " + computersNumber);
            break; // The game is over; the user has won.
        }
        if (guessCount == 6) {
            TextIO.putln("You didn't get the number in 6 guesses.");
            TextIO.putln("You lose. My number was " + computersNumber);
            break; // The game is over; the user has lost.
        }
        // If we get to this point, the game continues.
        // Tell the user if the guess was too high or too low.
        if (usersGuess < computersNumber)
            TextIO.put("That's too low. Try again: ");
        else if (usersGuess > computersNumber)
            TextIO.put("That's too high. Try again: ");
    }
    TextIO.putln();
} // end of playGame()

} // end of class GuessingGame

```

Take some time to read the program carefully and figure out how it works. And try to convince yourself that even in this relatively simple case, breaking up the program into two methods makes the program easier to understand and probably made it easier to write each piece.

4.2.4 Member Variables

A class can include other things besides subroutines. In particular, it can also include variable declarations. Of course, you can declare variables **inside** subroutines. Those are called *local variables*. However, you can also have variables that are not part of any subroutine. To

distinguish such variables from local variables, we call them *member variables*, since they are members of a class.

Just as with subroutines, member variables can be either static or non-static. In this chapter, we'll stick to static variables. A static member variable belongs to the class itself, and it exists as long as the class exists. Memory is allocated for the variable when the class is first loaded by the Java interpreter. Any assignment statement that assigns a value to the variable changes the content of that memory, no matter where that assignment statement is located in the program. Any time the variable is used in an expression, the value is fetched from that same memory, no matter where the expression is located in the program. This means that the value of a static member variable can be set in one subroutine and used in another subroutine. Static member variables are “shared” by all the static subroutines in the class. A local variable in a subroutine, on the other hand, exists only while that subroutine is being executed, and is completely inaccessible from outside that one subroutine.

The declaration of a member variable looks just like the declaration of a local variable except for two things: The member variable is declared outside any subroutine (although it still has to be inside a class), and the declaration can be marked with modifiers such as **static**, **public**, and **private**. Since we are only working with static member variables for now, every declaration of a member variable in this chapter will include the modifier **static**. They might also be marked as **public** or **private**. For example:

```
static String userName;  
public static int numberOfPlayers;  
private static double velocity, time;
```

A static member variable that is not declared to be **private** can be accessed from outside the class where it is defined, as well as inside. When it is used in some other class, it must be referred to with a compound identifier of the form `<class-name>.<variable-name>`. For example, the *System* class contains the public static member variable named `out`, and you use this variable in your own classes by referring to `System.out`. If `numberOfPlayers` is a public static member variable in a class named *Poker*, then subroutines in the *Poker* class would refer to it simply as `numberOfPlayers`, while subroutines in another class would refer to it as `Poker.numberOfPlayers`.

As an example, let's add a static member variable to the *GuessingGame* class that we wrote earlier in this section. This variable will be used to keep track of how many games the user wins. We'll call the variable `gamesWon` and declare it with the statement “`static int gamesWon;`”. In the `playGame()` routine, we add 1 to `gamesWon` if the user wins the game. At the end of the `main()` routine, we print out the value of `gamesWon`. It would be impossible to do the same thing with a local variable, since we need access to the same variable from both subroutines.

When you declare a local variable in a subroutine, you have to assign a value to that variable before you can do anything with it. Member variables, on the other hand are automatically initialized with a default value. For numeric variables, the default value is zero. For **boolean** variables, the default is **false**. And for **char** variables, it's the unprintable character that has Unicode code number zero. (For objects, such as *Strings*, the default initial value is a special value called **null**, which we won't encounter officially until later.)

Since it is of type **int**, the static member variable `gamesWon` automatically gets assigned an initial value of zero. This happens to be the correct initial value for a variable that is being used as a counter. You can, of course, assign a different value to the variable at the beginning of the `main()` routine if you are not satisfied with the default initial value.

Here's a revised version of `GuessingGame.java` that includes the `gamesWon` variable. The changes from the above version are shown in *italic*:

```
public class GuessingGame2 {

    static int gamesWon;          // The number of games won by
                                // the user.

    public static void main(String[] args) {
        gamesWon = 0; // This is actually redundant, since 0 is
                       // the default initial value.
        TextIO.putln("Let's play a game. I'll pick a number between");
        TextIO.putln("1 and 100, and you try to guess it.");
        boolean playAgain;
        do {
            playGame(); // call subroutine to play one game
            TextIO.put("Would you like to play again? ");
            playAgain = TextIO.getlnBoolean();
        } while (playAgain);
        TextIO.putln();
        TextIO.putln("You won " + gamesWon + " games.");
        TextIO.putln("Thanks for playing. Goodbye.");
    } // end of main()

    static void playGame() {
        int computersNumber; // A random number picked by the computer.
        int usersGuess;      // A number entered by user as a guess.
        int guessCount;      // Number of guesses the user has made.
        computersNumber = (int)(100 * Math.random()) + 1;
                       // The value assigned to computersNumber is a randomly
                       // chosen integer between 1 and 100, inclusive.
        guessCount = 0;
        TextIO.putln();
        TextIO.put("What is your first guess? ");
        while (true) {
            usersGuess = TextIO.getInt(); // Get the user's guess.
            guessCount++;
            if (usersGuess == computersNumber) {
                TextIO.putln("You got it in " + guessCount
                             + " guesses! My number was " + computersNumber);
                gamesWon++; // Count this game by incrementing gamesWon.
                break;      // The game is over; the user has won.
            }
            if (guessCount == 6) {
                TextIO.putln("You didn't get the number in 6 guesses.");
                TextIO.putln("You lose. My number was " + computersNumber);
                break; // The game is over; the user has lost.
            }
            // If we get to this point, the game continues.
            // Tell the user if the guess was too high or too low.
            if (usersGuess < computersNumber)
                TextIO.put("That's too low. Try again: ");
            else if (usersGuess > computersNumber)
                TextIO.put("That's too high. Try again: ");
        }
    }
}
```

```

        TextIO.putln();
    } // end of playGame()

} // end of class GuessingGame2

```

4.3 Parameters

IF A SUBROUTINE IS A BLACK BOX, then a parameter provides a mechanism for passing information from the outside world into the box. Parameters are part of the interface of a subroutine. They allow you to customize the behavior of a subroutine to adapt it to a particular situation.

As an analogy, consider a thermostat—a black box whose task it is to keep your house at a certain temperature. The thermostat has a parameter, namely the dial that is used to set the desired temperature. The thermostat always performs the same task: maintaining a constant temperature. However, the exact task that it performs—that is, **which** temperature it maintains—is customized by the setting on its dial.

4.3.1 Using Parameters

As an example, let's go back to the “ $3N+1$ ” problem that was discussed in Subsection 3.2.2. (Recall that a $3N+1$ sequence is computed according to the rule, “if N is odd, multiply by 3 and add 1; if N is even, divide by 2; continue until N is equal to 1.” For example, starting from $N=3$ we get the sequence: 3, 10, 5, 16, 8, 4, 2, 1.) Suppose that we want to write a subroutine to print out such sequences. The subroutine will always perform the same task: Print out a $3N+1$ sequence. But the exact sequence it prints out depends on the starting value of N . So, the starting value of N would be a parameter to the subroutine. The subroutine could be written like this:

```

/**
 * This subroutine prints a 3N+1 sequence to standard output, using
 * startingValue as the initial value of N. It also prints the number
 * of terms in the sequence. The value of the parameter, startingValue,
 * must be a positive integer.
 */

static void print3NSequence(int startingValue) {

    int N;          // One of the terms in the sequence.
    int count;      // The number of terms.

    N = startingValue; // The first term is whatever value
                       // is passed to the subroutine as
                       // a parameter.

    int count = 1; // We have one term, the starting value, so far.

    System.out.println("The 3N+1 sequence starting from " + N);
    System.out.println();
    System.out.println(N); // print initial term of sequence

    while (N > 1) {
        if (N % 2 == 1) // is N odd?
            N = 3 * N + 1;
        else
            N = N / 2;
    }
}

```

```

        count++;    // count this term
        System.out.println(N); // print this term
    }

    System.out.println();
    System.out.println("There were " + count + " terms in the sequence.");
} // end print3NSequence

```

The parameter list of this subroutine, “(int startingValue)”, specifies that the subroutine has one parameter, of type **int**. Within the body of the subroutine, the parameter name can be used in the same way as a variable name. However, the parameter gets its initial value from **outside** the subroutine. When the subroutine is called, a value must be provided for this parameter in the subroutine call statement. This value will be assigned to the parameter, **startingValue**, before the body of the subroutine is executed. For example, the subroutine could be called using the subroutine call statement “**print3NSequence(17);**”. When the computer executes this statement, the computer assigns the value 17 to **startingValue** and then executes the statements in the subroutine. This prints the $3N+1$ sequence starting from 17. If **K** is a variable of type **int**, then when the computer executes the subroutine call statement “**print3NSequence(K);**”, it will take the value of the variable **K**, assign that value to **startingValue**, and execute the body of the subroutine.

The class that contains **print3NSequence** can contain a **main()** routine (or other subroutines) that call **print3NSequence**. For example, here is a **main()** program that prints out $3N+1$ sequences for various starting values specified by the user:

```

public static void main(String[] args) {
    TextIO.putln("This program will print out 3N+1 sequences");
    TextIO.putln("for starting values that you specify.");
    TextIO.putln();
    int K; // Input from user; loop ends when K < 0.
    do {
        TextIO.putln("Enter a starting value;")
        TextIO.put("To end the program, enter 0: ");
        K = TextIO.getInt(); // Get starting value from user.
        if (K > 0) // Print sequence, but only if K is > 0.
            print3NSequence(K);
    } while (K > 0); // Continue only if K > 0.
} // end main

```

Remember that before you can use this program, the definitions of **main** and of **print3NSequence** must both be wrapped inside a class definition.

4.3.2 Formal and Actual Parameters

Note that the term “parameter” is used to refer to two different, but related, concepts. There are parameters that are used in the definitions of subroutines, such as **startingValue** in the above example. And there are parameters that are used in subroutine call statements, such as the **K** in the statement “**print3NSequence(K);**”. Parameters in a subroutine definition are called **formal parameters** or **dummy parameters**. The parameters that are passed to a subroutine when it is called are called **actual parameters** or **arguments**. When a subroutine is called, the actual parameters in the subroutine call statement are evaluated and the values are assigned to the formal parameters in the subroutine’s definition. Then the body of the subroutine is executed.

A formal parameter must be a **name**, that is, a simple identifier. A formal parameter is very much like a variable, and—like a variable—it has a specified type such as **int**, **boolean**, or *String*. An actual parameter is a **value**, and so it can be specified by any expression, provided that the expression computes a value of the correct type. The type of the actual parameter must be one that could legally be assigned to the formal parameter with an assignment statement. For example, if the formal parameter is of type **double**, then it would be legal to pass an **int** as the actual parameter since **ints** can legally be assigned to **doubles**. When you call a subroutine, you must provide one actual parameter for each formal parameter in the subroutine’s definition. Consider, for example, a subroutine

```
static void doTask(int N, double x, boolean test) {
    // statements to perform the task go here
}
```

This subroutine might be called with the statement

```
doTask(17, Math.sqrt(z+1), z >= 10);
```

When the computer executes this statement, it has essentially the same effect as the block of statements:

```
{
    int N;          // Allocate memory locations for the formal parameters.
    double x;
    boolean test;
    N = 17;          // Assign 17 to the first formal parameter, N.
    x = Math.sqrt(z+1); // Compute Math.sqrt(z+1), and assign it to
                      // the second formal parameter, x.
    test = (z >= 10); // Evaluate "z >= 10" and assign the resulting
                      // true/false value to the third formal
                      // parameter, test.
    // statements to perform the task go here
}
```

(There are a few technical differences between this and “doTask(17,Math.sqrt(z+1),z>=10);”—besides the amount of typing—because of questions about scope of variables and what happens when several variables or parameters have the same name.)

Beginning programming students often find parameters to be surprisingly confusing. Calling a subroutine that already exists is not a problem—the idea of providing information to the subroutine in a parameter is clear enough. Writing the subroutine definition is another matter. A common mistake is to assign values to the formal parameters at the beginning of the subroutine, or to ask the user to input their values. This represents a fundamental misunderstanding. When the statements in the subroutine are executed, the formal parameters will already have values. The values come from the subroutine call statement. Remember that a subroutine is not independent. It is called by some other routine, and it is the calling routine’s responsibility to provide appropriate values for the parameters.

4.3.3 Overloading

In order to call a subroutine legally, you need to know its name, you need to know how many formal parameters it has, and you need to know the type of each parameter. This information is called the subroutine’s *signature*. The signature of the subroutine `doTask`, used as an example above, can be expressed as as: `doTask(int,double,boolean)`. Note that the signature does

not include the names of the parameters; in fact, if you just want to **use** the subroutine, you don't even need to know what the formal parameter names are, so the names are not part of the interface.

Java is somewhat unusual in that it allows two different subroutines in the same class to have the same name, provided that their signatures are different. (The language C++ on which Java is based also has this feature.) When this happens, we say that the name of the subroutine is *overloaded* because it has several different meanings. The computer doesn't get the subroutines mixed up. It can tell which one you want to call by the number and types of the actual parameters that you provide in the subroutine call statement. You have already seen overloading used in the *TextIO* class. This class includes many different methods named `putln`, for example. These methods all have different signatures, such as:

<code>putln(int)</code>	<code>putln(double)</code>
<code>putln(String)</code>	<code>putln(char)</code>
<code>putln(boolean)</code>	<code>putln()</code>

The computer knows which of these subroutines you want to use based on the type of the actual parameter that you provide. `TextIO.putln(17)` calls the subroutine with signature `putln(int)`, while `TextIO.putln("Hello")` calls the subroutine with signature `putln(String)`. Of course all these different subroutines are semantically related, which is why it is acceptable programming style to use the same name for them all. But as far as the computer is concerned, printing out an **int** is very different from printing out a *String*, which is different from printing out a **boolean**, and so forth—so that each of these operations requires a different method.

Note, by the way, that the signature does **not** include the subroutine's return type. It is illegal to have two subroutines in the same class that have the same signature but that have different return types. For example, it would be a syntax error for a class to contain two methods defined as:

```
int    getln() { ... }
double getln() { ... }
```

So it should be no surprise that in the *TextIO* class, the methods for reading different types are not all named `getln()`. In a given class, there can only be one routine that has the name `getln` and has no parameters. So, the input routines in *TextIO* are distinguished by having different names, such as `getlnInt()` and `getlnDouble()`.

Java 5.0 introduced another complication: It is possible to have a single subroutine that takes a variable number of actual parameters. You have already used subroutines that do this—the formatted output routines `System.out.printf` and `TextIO.putf`. When you call these subroutines, the number of parameters in the subroutine call can be arbitrarily large, so it would be impossible to have different subroutines to handle each case. Unfortunately, writing the definition of such a subroutine requires some knowledge of arrays, which will not be covered until Chapter 7. When we get to that chapter, you'll learn how to write subroutines with a variable number of parameters. For now, we will ignore this complication.

4.3.4 Subroutine Examples

Let's do a few examples of writing small subroutines to perform assigned tasks. Of course, this is only one side of programming with subroutines. The task performed by a subroutine is always a subtask in a larger program. The art of designing those programs—of deciding how to

break them up into subtasks—is the other side of programming with subroutines. We’ll return to the question of program design in Section 4.6.

As a first example, let’s write a subroutine to compute and print out all the divisors of a given positive integer. The integer will be a parameter to the subroutine. Remember that the syntax of any subroutine is:

```

    <modifiers> <return-type> <subroutine-name> ( <parameter-list> ) {
        <statements>
    }

```

Writing a subroutine always means filling out this format. In this case, the statement of the problem tells us that there is one parameter, of type **int**, and it tells us what the statements in the body of the subroutine should do. Since we are only working with static subroutines for now, we’ll need to use **static** as a modifier. We could add an access modifier (**public** or **private**), but in the absence of any instructions, I’ll leave it out. Since we are not told to return a value, the return type is **void**. Since no names are specified, we’ll have to make up names for the formal parameter and for the subroutine itself. I’ll use **N** for the parameter and **printDivisors** for the subroutine name. The subroutine will look like

```

    static void printDivisors( int N ) {
        <statements>
    }

```

and all we have left to do is to write the statements that make up the body of the routine. This is not difficult. Just remember that you have to write the body assuming that **N** already has a value! The algorithm is: “For each possible divisor **D** in the range from 1 to **N**, if **D** evenly divides **N**, then print **D**.” Written in Java, this becomes:

```

/**
 * Print all the divisors of N.
 * We assume that N is a positive integer.
 */
static void printDivisors( int N ) {
    int D;    // One of the possible divisors of N.
    System.out.println("The divisors of " + N + " are:");
    for ( D = 1; D <= N; D++ ) {
        if ( N % D == 0 )
            System.out.println(D);
    }
}

```

I’ve added a comment before the subroutine definition indicating the contract of the subroutine—that is, what it does and what assumptions it makes. The contract includes the assumption that **N** is a positive integer. It is up to the caller of the subroutine to make sure that this assumption is satisfied.

As a second short example, consider the problem: Write a subroutine named **printRow**. It should have a parameter **ch** of type **char** and a parameter **N** of type **int**. The subroutine should print out a line of text containing **N** copies of the character **ch**.

Here, we are told the name of the subroutine and the names of the two parameters, so we don’t have much choice about the first line of the subroutine definition. The task in this case is pretty simple, so the body of the subroutine is easy to write. The complete subroutine is given by

```

/**
 * Write one line of output containing N copies of the
 * character ch.  If N <= 0, an empty line is output.
 */

static void printRow( char ch, int N ) {
    int i; // Loop-control variable for counting off the copies.
    for ( i = 1; i <= N; i++ ) {
        System.out.print( ch );
    }
    System.out.println();
}

```

Note that in this case, the contract makes no assumption about *N*, but it makes it clear what will happen in all cases, including the unexpected case that $N < 0$.

Finally, let's do an example that shows how one subroutine can build on another. Let's write a subroutine that takes a *String* as a parameter. For each character in the string, it will print a line of output containing 25 copies of that character. It should use the `printRow()` subroutine to produce the output.

Again, we get to choose a name for the subroutine and a name for the parameter. I'll call the subroutine `printRowsFromString` and the parameter `str`. The algorithm is pretty clear: For each position *i* in the string `str`, call `printRow(str.charAt(i),25)` to print one line of the output. So, we get:

```

/**
 * For each character in str, write a line of output
 * containing 25 copies of that character.
 */

static void printRowsFromString( String str ) {
    int i; // Loop-control variable for counting off the chars.
    for ( i = 0; i < str.length(); i++ ) {
        printRow( str.charAt(i), 25 );
    }
}

```

We could use `printRowsFromString` in a `main()` routine such as

```

public static void main(String[] args) {
    String inputLine; // Line of text input by user.
    TextIO.put("Enter a line of text: ");
    inputLine = TextIO.getln();
    TextIO.putln();
    printRowsFromString( inputLine );
}

```

Of course, the three routines, `main()`, `printRowsFromString()`, and `printRow()`, would have to be collected together inside the same class. The program is rather useless, but it does demonstrate the use of subroutines. You'll find the program in the file *RowsOfChars.java*, if you want to take a look.

4.3.5 Throwing Exceptions

I have been talking about the “contract” of a subroutine. The contract says what the subroutine will do, provided that the caller of the subroutine provides acceptable values for subroutine's

parameters. The question arises, though, what should the subroutine do when the caller violates the contract by providing bad parameter values?

We’ve already seen that some subroutines respond to bad parameter values by throwing exceptions. (See Section 3.7.) For example, the contract of the built-in subroutine `Double.parseDouble` says that the parameter should be a string representation of a number of type **double**; if this is true, then the subroutine will convert the string into the equivalent numeric value. If the caller violates the contract by passing an invalid string as the actual parameter, the subroutine responds by throwing an exception of type *NumberFormatException*.

Many subroutines throw *IllegalArgumentException* in response to bad parameter values. You might want to take this response in your own subroutines. This can be done with a **throw statement**. An exception is an object, and in order to throw an exception, you must create an exception object. You won’t officially learn how to do this until Chapter 5, but for now, you can use the following syntax for a **throw** statement that throws an *IllegalArgumentException*:

```
throw new IllegalArgumentException( <error-message> );
```

where *<error-message>* is a string that describes the error that has been detected. (The word “new” in this statement is what creates the object.) To use this statement in a subroutine, you would check whether the values of the parameters are legal. If not, you would throw the exception. For example, consider the `print3NSequence` subroutine from the beginning of this section. The parameter of `print3NSequence` is supposed to be a positive integer. We can modify the subroutine definition to make it throw an exception when this condition is violated:

```
static void print3NSequence(int startingValue) {
    if (startingValue <= 0) // The contract is violated!
        throw new IllegalArgumentException( "Starting value must be positive." );
    .
    . // (The rest of the subroutine is the same as before.)
    .
}
```

If the start value is bad, the computer executes the **throw** statement. This will immediately terminate the subroutine, without executing the rest of the body of the subroutine. Furthermore, the program as a whole will crash unless the exception is “caught” and handled elsewhere in the program by a `try...catch` statement, as discussed in Section 3.7.

4.3.6 Global and Local Variables

I’ll finish this section on parameters by noting that we now have three different sorts of variables that can be used inside a subroutine: local variables declared in the subroutine, formal parameter names, and static member variables that are declared outside the subroutine but inside the same class as the subroutine.

Local variables have no connection to the outside world; they are purely part of the internal working of the subroutine. Parameters are used to “drop” values into the subroutine when it is called, but once the subroutine starts executing, parameters act much like local variables. Changes made inside a subroutine to a formal parameter have no effect on the rest of the program (at least if the type of the parameter is one of the primitive types—things are more complicated in the case of objects, as we’ll see later).

Things are different when a subroutine uses a variable that is defined outside the subroutine. That variable exists independently of the subroutine, and it is accessible to other parts of the program, as well as to the subroutine. Such a variable is said to be **global** to the subroutine,

as opposed to the local variables defined inside the subroutine. The scope of a global variable includes the entire class in which it is defined. Changes made to a global variable can have effects that extend outside the subroutine where the changes are made. You’ve seen how this works in the last example in the previous section, where the value of the global variable, `gamesWon`, is computed inside a subroutine and is used in the `main()` routine.

It’s not always bad to use global variables in subroutines, but you should realize that the global variable then has to be considered part of the subroutine’s interface. The subroutine uses the global variable to communicate with the rest of the program. This is a kind of sneaky, back-door communication that is less visible than communication done through parameters, and it risks violating the rule that the interface of a black box should be straightforward and easy to understand. So before you use a global variable in a subroutine, you should consider whether it’s really necessary.

I don’t advise you to take an absolute stand against using global variables inside subroutines. There is at least one good reason to do it: If you think of the class as a whole as being a kind of black box, it can be very reasonable to let the subroutines inside that box be a little sneaky about communicating with each other, if that will make the class as a whole look simpler from the outside.

4.4 Return Values

A SUBROUTINE THAT RETURNS A VALUE is called a *function*. A given function can only return a value of a specified type, called the *return type* of the function. A function call generally occurs in a position where the computer is expecting to find a value, such as the right side of an assignment statement, as an actual parameter in a subroutine call, or in the middle of some larger expression. A boolean-valued function can even be used as the test condition in an `if`, `while`, `for` or `do..while` statement.

(It is also legal to use a function call as a stand-alone statement, just as if it were a regular subroutine. In this case, the computer ignores the value computed by the subroutine. Sometimes this makes sense. For example, the function `TextIO.getln()`, with a return type of *String*, reads and returns a line of input typed in by the user. Usually, the line that is returned is assigned to a variable to be used later in the program, as in the statement “`name = TextIO.getln();`”. However, this function is also useful as a subroutine call statement “`TextIO.getln();`”, which still reads all input up to and including the next carriage return. Since the return value is not assigned to a variable or used in an expression, it is simply discarded. So, the effect of the subroutine call is to read **and discard** some input. Sometimes, discarding unwanted input is exactly what you need to do.)

4.4.1 The return statement

You’ve already seen how functions such as `Math.sqrt()` and `TextIO.getInt()` can be used. What you haven’t seen is how to write functions of your own. A function takes the same form as a regular subroutine, except that you have to specify the value that is to be returned by the subroutine. This is done with a *return statement*, which has the following syntax:

```
return <expression> ;
```

Such a **return** statement can only occur inside the definition of a function, and the type of the *<expression>* must match the return type that was specified for the function. (More exactly, it must be legal to assign the expression to a variable whose type is specified by the

return type.) When the computer executes this **return** statement, it evaluates the expression, terminates execution of the function, and uses the value of the expression as the returned value of the function.

For example, consider the function definition

```
static double pythagoras(double x, double y) {
    // Computes the length of the hypotenuse of a right
    // triangle, where the sides of the triangle are x and y.
    return Math.sqrt( x*x + y*y );
}
```

Suppose the computer executes the statement “`totalLength = 17 + pythagoras(12,5);`”. When it gets to the term `pythagoras(12,5)`, it assigns the actual parameters 12 and 5 to the formal parameters `x` and `y` in the function. In the body of the function, it evaluates `Math.sqrt(12.0*12.0 + 5.0*5.0)`, which works out to 13.0. This value is “returned” by the function, so the 13.0 essentially replaces the function call in the statement “`totalLength = 17 + pythagoras(12,5);`”. The return value is added to 17, and the result, 30.0, is stored in the variable, `totalLength`. The effect is the same as if the statement had been “`totalLength = 17 + 13.0;`”.

Note that a **return** statement does not have to be the last statement in the function definition. At any point in the function where you know the value that you want to return, you can return it. Returning a value will end the function immediately, skipping any subsequent statements in the function. However, it must be the case that the function definitely does return some value, no matter what path the execution of the function takes through the code.

You can use a **return** statement inside an ordinary subroutine, one with declared return type “`void`”. Since a void subroutine does not return a value, the **return** statement does not include an expression; it simply takes the form “**return;**”. The effect of this statement is to terminate execution of the subroutine and return control back to the point in the program from which the subroutine was called. This can be convenient if you want to terminate execution somewhere in the middle of the subroutine, but **return** statements are optional in non-function subroutines. In a function, on the other hand, a return statement, with expression, is always required.

4.4.2 Function Examples

Here is a very simple function that could be used in a program to compute $3N+1$ sequences. (The $3N+1$ sequence problem is one we’ve looked at several times already, including in the previous section). Given one term in a $3N+1$ sequence, this function computes the next term of the sequence:

```
static int nextN(int currentN) {
    if (currentN % 2 == 1)    // test if current N is odd
        return 3*currentN + 1; // if so, return this value
    else
        return currentN / 2;   // if not, return this instead
}
```

This function has two **return** statements. Exactly one of the two **return** statements is executed to give the value of the function. Some people prefer to use a single **return** statement at the very end of the function when possible. This allows the reader to find the **return** statement easily. You might choose to write `nextN()` like this, for example:

```

static int nextN(int currentN) {
    int answer; // answer will be the value returned
    if (currentN % 2 == 1) // test if current N is odd
        answer = 3*currentN+1; // if so, this is the answer
    else
        answer = currentN / 2; // if not, this is the answer
    return answer; // (Don't forget to return the answer!)
}

```

Here is a subroutine that uses this `nextN` function. In this case, the improvement from the version of this subroutine in Section 4.3 is not great, but if `nextN()` were a long function that performed a complex computation, then it would make a lot of sense to hide that complexity inside a function:

```

static void print3NSequence(int startingValue) {

    int N; // One of the terms in the sequence.
    int count; // The number of terms found.

    N = startingValue; // Start the sequence with startingValue.
    count = 1;

    TextIO.putln("The 3N+1 sequence starting from " + N);
    TextIO.putln();
    TextIO.putln(N); // print initial term of sequence

    while (N > 1) {
        N = nextN(N); // Compute next term, using the function nextN.
        count++; // Count this term.
        TextIO.putln(N); // Print this term.
    }

    TextIO.putln();
    TextIO.putln("There were " + count + " terms in the sequence.");
}

```

* * *

Here are a few more examples of functions. The first one computes a letter grade corresponding to a given numerical grade, on a typical grading scale:

```

/**
 * Returns the letter grade corresponding to the numerical
 * grade that is passed to this function as a parameter.
 */

static char letterGrade(int numGrade) {
    if (numGrade >= 90)
        return 'A'; // 90 or above gets an A
    else if (numGrade >= 80)
        return 'B'; // 80 to 89 gets a B
    else if (numGrade >= 65)
        return 'C'; // 65 to 79 gets a C
    else if (numGrade >= 50)
        return 'D'; // 50 to 64 gets a D
    else

```

```

        return 'F';    // anything else gets an F
    } // end of function letterGrade

```

The type of the return value of `letterGrade()` is **char**. Functions can return values of any type at all. Here's a function whose return value is of type **boolean**. It demonstrates some interesting programming points, so you should read the comments:

```

/**
 * The function returns true if N is a prime number. A prime number
 * is an integer greater than 1 that is not divisible by any positive
 * integer, except itself and 1. If N has any divisor, D, in the range
 * 1 < D < N, then it has a divisor in the range 2 to Math.sqrt(N), namely
 * either D itself or N/D. So we only test possible divisors from 2 to
 * Math.sqrt(N).
 */
static boolean isPrime(int N) {
    int divisor; // A number we will test to see whether it evenly divides N.
    if (N <= 1)
        return false; // No number <= 1 is a prime.

    int maxToTry; // The largest divisor that we need to test.
    maxToTry = (int)Math.sqrt(N);
    // We will try to divide N by numbers between 2 and maxToTry.
    // If N is not evenly divisible by any of these numbers, then
    // N is prime. (Note that since Math.sqrt(N) is defined to
    // return a value of type double, the value must be typecast
    // to type int before it can be assigned to maxToTry.)

    for (divisor = 2; divisor <= maxToTry; divisor++) {
        if ( N % divisor == 0 ) // Test if divisor evenly divides N.
            return false;      // If so, we know N is not prime.
                                // No need to continue testing!
    }

    // If we get to this point, N must be prime. Otherwise,
    // the function would already have been terminated by
    // a return statement in the previous loop.

    return true; // Yes, N is prime.
} // end of function isPrime

```

Finally, here is a function with return type *String*. This function has a *String* as parameter. The returned value is a reversed copy of the parameter. For example, the reverse of “Hello World” is “dlroW olleH”. The algorithm for computing the reverse of a string, `str`, is to start with an empty string and then to append each character from `str`, starting from the last character of `str` and working backwards to the first:

```

static String reverse(String str) {
    String copy; // The reversed copy.
    int i;       // One of the positions in str,
                //           from str.length() - 1 down to 0.
    copy = "";   // Start with an empty string.
    for ( i = str.length() - 1; i >= 0; i-- ) {

```

```

        // Append i-th char of str to copy.
        copy = copy + str.charAt(i);
    }
    return copy;
}

```

A *palindrome* is a string that reads the same backwards and forwards, such as “radar”. The `reverse()` function could be used to check whether a string, `word`, is a palindrome by testing “`if (word.equals(reverse(word)))`”.

By the way, a typical beginner’s error in writing functions is to print out the answer, instead of returning it. This represents a fundamental misunderstanding. The task of a function is to compute a value and return it to the point in the program where the function was called. That’s where the value is used. Maybe it will be printed out. Maybe it will be assigned to a variable. Maybe it will be used in an expression. But it’s not for the function to decide.

4.4.3 3N+1 Revisited

I’ll finish this section with a complete new version of the 3N+1 program. This will give me a chance to show the function `nextN()`, which was defined above, used in a complete program. I’ll also take the opportunity to improve the program by getting it to print the terms of the sequence in columns, with five terms on each line. This will make the output more presentable. This idea is this: Keep track of how many terms have been printed on the current line; when that number gets up to 5, start a new line of output. To make the terms line up into neat columns, I use formatted output.

```

/**
 * A program that computes and displays several 3N+1 sequences. Starting
 * values for the sequences are input by the user. Terms in the sequence
 * are printed in columns, with five terms on each line of output.
 * After a sequence has been displayed, the number of terms in that
 * sequence is reported to the user.
 */

public class ThreeN2 {

    public static void main(String[] args) {

        TextIO.putln("This program will print out 3N+1 sequences");
        TextIO.putln("for starting values that you specify.");
        TextIO.putln();

        int K;    // Starting point for sequence, specified by the user.
        do {
            TextIO.putln("Enter a starting value;");
            TextIO.put("To end the program, enter 0: ");
            K = TextIO.getInt();    // get starting value from user
            if (K > 0)                // print sequence, but only if K is > 0
                print3NSequence(K);
        } while (K > 0);            // continue only if K > 0

    } // end main

}

/**
 * print3NSequence prints a 3N+1 sequence to standard output, using

```

```

* startingValue as the initial value of N. It also prints the number
* of terms in the sequence. The value of the parameter, startingValue,
* must be a positive integer.
*/
static void print3NSequence(int startingValue) {

    int N;          // One of the terms in the sequence.
    int count;      // The number of terms found.
    int onLine;     // The number of terms that have been output
                   // so far on the current line.

    N = startingValue; // Start the sequence with startingValue;
    count = 1;         // We have one term so far.

    TextIO.putln("The 3N+1 sequence starting from " + N);
    TextIO.putln();
    TextIO.put(N, 8); // Print initial term, using 8 characters.
    onLine = 1;       // There's now 1 term on current output line.

    while (N > 1) {
        N = nextN(N); // compute next term
        count++;      // count this term
        if (onLine == 5) { // If current output line is full
            TextIO.putln(); // ...then output a carriage return
            onLine = 0;     // ...and note that there are no terms
                           // on the new line.
        }
        TextIO.printf("%8d", N); // Print this term in an 8-char column.
        onLine++;              // Add 1 to the number of terms on this line.
    }

    TextIO.putln(); // end current line of output
    TextIO.putln(); // and then add a blank line
    TextIO.putln("There were " + count + " terms in the sequence.");
} // end of Print3NSequence

/**
 * nextN computes and returns the next term in a 3N+1 sequence,
 * given that the current term is currentN.
 */
static int nextN(int currentN) {
    if (currentN % 2 == 1)
        return 3 * currentN + 1;
    else
        return currentN / 2;
} // end of nextN()

} // end of class ThreeN2

```

You should read this program carefully and try to understand how it works. (Try using 27 for the starting value!)

4.5 APIs, Packages, and Javadoc

AS COMPUTERS AND THEIR USER INTERFACES have become easier to use, they have also become more complex for programmers to deal with. You can write programs for a simple console-style user interface using just a few subroutines that write output to the console and read the user's typed replies. A modern graphical user interface, with windows, buttons, scroll bars, menus, text-input boxes, and so on, might make things easier for the user, but it forces the programmer to cope with a hugely expanded array of possibilities. The programmer sees this increased complexity in the form of great numbers of subroutines that are provided for managing the user interface, as well as for other purposes.

4.5.1 Toolboxes

Someone who wants to program for Macintosh computers—and to produce programs that look and behave the way users expect them to—must deal with the Macintosh Toolbox, a collection of well over a thousand different subroutines. There are routines for opening and closing windows, for drawing geometric figures and text to windows, for adding buttons to windows, and for responding to mouse clicks on the window. There are other routines for creating menus and for reacting to user selections from menus. Aside from the user interface, there are routines for opening files and reading data from them, for communicating over a network, for sending output to a printer, for handling communication between programs, and in general for doing all the standard things that a computer has to do. Microsoft Windows provides its own set of subroutines for programmers to use, and they are quite a bit different from the subroutines used on the Mac. Linux has several different GUI toolboxes for the programmer to choose from.

The analogy of a “toolbox” is a good one to keep in mind. Every programming project involves a mixture of innovation and reuse of existing tools. A programmer is given a set of tools to work with, starting with the set of basic tools that are built into the language: things like variables, assignment statements, if statements, and loops. To these, the programmer can add existing toolboxes full of routines that have already been written for performing certain tasks. These tools, if they are well-designed, can be used as true black boxes: They can be called to perform their assigned tasks without worrying about the particular steps they go through to accomplish those tasks. The innovative part of programming is to take all these tools and apply them to some particular project or problem (word-processing, keeping track of bank accounts, processing image data from a space probe, Web browsing, computer games, . . .). This is called *applications programming*.

A software toolbox is a kind of black box, and it presents a certain interface to the programmer. This interface is a specification of what routines are in the toolbox, what parameters they use, and what tasks they perform. This information constitutes the **API**, or **Applications Programming Interface**, associated with the toolbox. The Macintosh API is a specification of all the routines available in the Macintosh Toolbox. A company that makes some hardware device—say a card for connecting a computer to a network—might publish an API for that device consisting of a list of routines that programmers can call in order to communicate with and control the device. Scientists who write a set of routines for doing some kind of complex computation—such as solving “differential equations,” say—would provide an API to allow others to use those routines without understanding the details of the computations they perform.

The Java programming language is supplemented by a large, standard API. You've seen part of this API already, in the form of mathematical subroutines such as `Math.sqrt()`, the *String* data type and its associated routines, and the `System.out.print()` routines. The standard Java API includes routines for working with graphical user interfaces, for network communication, for reading and writing files, and more. It's tempting to think of these routines as being built into the Java language, but they are technically subroutines that have been written and made available for use in Java programs.

Java is platform-independent. That is, the same program can run on platforms as diverse as Macintosh, Windows, Linux, and others. The same Java API must work on all these platforms. But notice that it is the **interface** that is platform-independent; the **implementation** varies from one platform to another. A Java system on a particular computer includes implementations of all the standard API routines. A Java program includes only **calls** to those routines. When the Java interpreter executes a program and encounters a call to one of the standard routines, it will pull up and execute the implementation of that routine which is appropriate for the particular platform on which it is running. This is a very powerful idea. It means that you only need to learn one API to program for a wide variety of platforms.

4.5.2 Java's Standard Packages

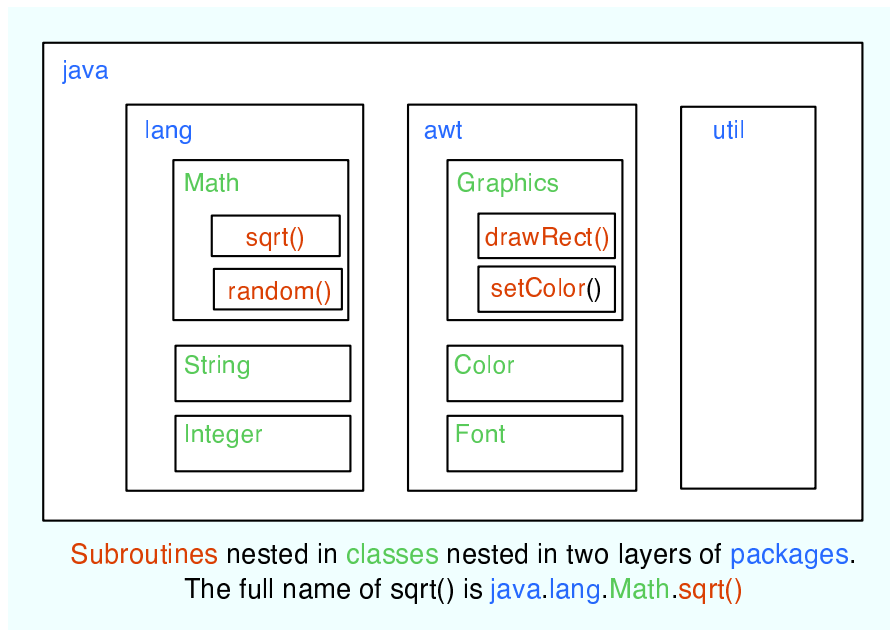
Like all subroutines in Java, the routines in the standard API are grouped into classes. To provide larger-scale organization, classes in Java can be grouped into **packages**, which were introduced briefly in Subsection 2.6.4. You can have even higher levels of grouping, since packages can also contain other packages. In fact, the entire standard Java API is implemented in several packages. One of these, which is named "java", contains several non-GUI packages as well as the original AWT graphics user interface classes. Another package, "javax", was added in Java version 1.2 and contains the classes used by the Swing graphical user interface and other additions to the API.

A package can contain both classes and other packages. A package that is contained in another package is sometimes called a "sub-package." Both the java package and the javax package contain sub-packages. One of the sub-packages of java, for example, is called "awt". Since `awt` is contained within `java`, its full name is actually `java.awt`. This package contains classes that represent GUI components such as buttons and menus in the AWT, the older of the two Java GUI toolboxes, which is no longer widely used. However, `java.awt` also contains a number of classes that form the foundation for all GUI programming, such as the `Graphics` class which provides routines for drawing on the screen, the `Color` class which represents colors, and the `Font` class which represents the fonts that are used to display characters on the screen. Since these classes are contained in the package `java.awt`, their full names are actually `java.awt.Graphics`, `java.awt.Color`, and `java.awt.Font`. (I hope that by now you've gotten the hang of how this naming thing works in Java.) Similarly, `javax` contains a sub-package named `javax.swing`, which includes such classes as `javax.swing.JButton`, `javax.swing.JMenu`, and `javax.swing.JFrame`. The GUI classes in `javax.swing`, together with the foundational classes in `java.awt`, are all part of the API that makes it possible to program graphical user interfaces in Java.

The `java` package includes several other sub-packages, such as `java.io`, which provides facilities for input/output, `java.net`, which deals with network communication, and `java.util`, which provides a variety of "utility" classes. The most basic package is called `java.lang`. This package contains fundamental classes such as *String*, *Math*, *Integer*, and *Double*.

It might be helpful to look at a graphical representation of the levels of nesting in the

java package, its sub-packages, the classes in those sub-packages, and the subroutines in those classes. This is not a complete picture, since it shows only a very few of the many items in each element:



The official documentation for the standard Java 5.0 API lists 165 different packages, including sub-packages, and it lists 3278 classes in these packages. Many of these are rather obscure or very specialized, but you might want to browse through the documentation to see what is available. As I write this, the documentation for the complete API can be found at

<http://java.sun.com/j2se/1.5.0/docs/api/index.html>

Even an expert programmer won't be familiar with the entire API, or even a majority of it. In this book, you'll only encounter several dozen classes, and those will be sufficient for writing a wide variety of programs.

4.5.3 Using Classes from Packages

Let's say that you want to use the class `java.awt.Color` in a program that you are writing. Like any class, `java.awt.Color` is a type, which means that you can use it to declare variables and parameters and to specify the return type of a function. One way to do this is to use the full name of the class as the name of the type. For example, suppose that you want to declare a variable named `rectColor` of type `java.awt.Color`. You could say:

```
java.awt.Color rectColor;
```

This is just an ordinary variable declaration of the form “*<type-name> <variable-name>;*”. Of course, using the full name of every class can get tiresome, so Java makes it possible to avoid using the full name of a class by *importing* the class. If you put

```
import java.awt.Color;
```

at the beginning of a Java source code file, then, in the rest of the file, you can abbreviate the full name `java.awt.Color` to just the simple name of the class, `Color`. Note that the `import`

line comes at the start of a file and is not inside any class. Although it is sometimes referred to as a statement, it is more properly called an *import directive* since it is not a statement in the usual sense. Using this `import` directive would allow you to say

```
Color rectColor;
```

to declare the variable. Note that the only effect of the `import` directive is to allow you to use simple class names instead of full “package.class” names; you aren’t really importing anything substantial. If you leave out the `import` directive, you can still access the class—you just have to use its full name. There is a shortcut for importing all the classes from a given package. You can import all the classes from `java.awt` by saying

```
import java.awt.*;
```

The “*” is a *wildcard* that matches every class in the package. (However, it does not match sub-packages; you **cannot** import the entire contents of all the sub-packages of the `java` package by saying `import java.*`.)

Some programmers think that using a wildcard in an `import` statement is bad style, since it can make a large number of class names available that you are not going to use and might not even know about. They think it is better to explicitly import each individual class that you want to use. In my own programming, I often use wildcards to import all the classes from the most relevant packages, and use individual imports when I am using just one or two classes from a given package.

In fact, any Java program that uses a graphical user interface is likely to use many classes from the `java.awt` and `javax.swing` packages as well as from another package named `java.awt.event`, and I usually begin such programs with

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

A program that works with networking might include the line “`import java.net.*`,” while one that reads or writes files might use “`import java.io.*`.” (But when you start importing lots of packages in this way, you have to be careful about one thing: It’s possible for two classes that are in different packages to have the same name. For example, both the `java.awt` package and the `java.util` package contain classes named `List`. If you import both `java.awt.*` and `java.util.*`, the simple name `List` will be ambiguous. If you try to declare a variable of type `List`, you will get a compiler error message about an ambiguous class name. The solution is simple: Use the full name of the class, either `java.awt.List` or `java.util.List`. Another solution, of course, is to use `import` to import the individual classes you need, instead of importing entire packages.)

Because the package `java.lang` is so fundamental, all the classes in `java.lang` are **automatically** imported into every program. It’s as if every program began with the statement “`import java.lang.*`.” This is why we have been able to use the class name *String* instead of `java.lang.String`, and `Math.sqrt()` instead of `java.lang.Math.sqrt()`. It would still, however, be perfectly legal to use the longer forms of the names.

Programmers can create new packages. Suppose that you want some classes that you are writing to be in a package named `utilities`. Then the source code file that defines those classes must begin with the line

```
package utilities;
```

This would come even before any `import` directive in that file. Furthermore, as mentioned in Subsection 2.6.4, the source code file would be placed in a folder with the same name as the package. A class that is in a package automatically has access to other classes in the same package; that is, a class doesn't have to import the package in which it is defined.

In projects that define large numbers of classes, it makes sense to organize those classes into packages. It also makes sense for programmers to create new packages as toolboxes that provide functionality and APIs for dealing with areas not covered in the standard Java API. (And in fact such “toolmaking” programmers often have more prestige than the applications programmers who use their tools.)

However, I will not be creating any packages in this textbook. For the purposes of this book, you need to know about packages mainly so that you will be able to import the standard packages. These packages are always available to the programs that you write. You might wonder where the standard classes are actually located. Again, that can depend to some extent on the version of Java that you are using, but in the standard Java 5.0, they are stored in *jar files* in a subdirectory of the main Java installation directory. A jar (or “Java archive”) file is a single file that can contain many classes. Most of the standard classes can be found in a jar file named `classes.jar`. In fact, Java programs are generally distributed in the form of jar files, instead of as individual class files.

Although we won't be creating packages explicitly, **every** class is actually part of a package. If a class is not specifically placed in a package, then it is put in something called the *default package*, which has no name. All the examples that you see in this book are in the default package.

4.5.4 Javadoc

To use an API effectively, you need good documentation for it. The documentation for most Java APIs is prepared using a system called *Javadoc*. For example, this system is used to prepare the documentation for Java's standard packages. And almost everyone who creates a toolbox in Java publishes Javadoc documentation for it.

Javadoc documentation is prepared from special comments that are placed in the Java source code file. Recall that one type of Java comment begins with `/*` and ends with `*/`. A Javadoc comment takes the same form, but it begins with `/**` rather than simply `/*`. You have already seen comments of this form in some of the examples in this book, such as this subroutine from Section 4.3:

```
/**
 * This subroutine prints a 3N+1 sequence to standard output, using
 * startingValue as the initial value of N. It also prints the number
 * of terms in the sequence. The value of the parameter, startingValue,
 * must be a positive integer.
 */

static void print3NSequence(int startingValue) { ...
```

Note that the Javadoc comment is placed just **before** the subroutine that it is commenting on. This rule is always followed. You can have Javadoc comments for subroutines, for member variables, and for classes. The Javadoc comment always immediately precedes the thing it is commenting on.

Like any comment, a Javadoc comment is ignored by the computer when the file is compiled. But there is a tool called `javadoc` that reads Java source code files, extracts any Javadoc

comments that it finds, and creates a set of Web pages containing the comments in a nicely formatted, interlinked form. By default, `javadoc` will only collect information about **public** classes, subroutines, and member variables, but it allows the option of creating documentation for non-public things as well. If `javadoc` doesn't find any Javadoc comment for something, it will construct one, but the comment will contain only basic information such as the name and type of a member variable or the name, return type, and parameter list of a subroutine. This is **syntactic** information. To add information about semantics and pragmatics, you have to write a Javadoc comment.

As an example, you can look at the documentation Web page for *TextIO*. The documentation page was created by applying the `javadoc` tool to the source code file, *TextIO.java*. If you have downloaded the on-line version of this book, the documentation can be found in the `TextIO.Javadoc` directory, or you can find a link to it in the on-line version of this section.

In a Javadoc comment, the `*`'s at the start of each line are optional. The `javadoc` tool will remove them. In addition to normal text, the comment can contain certain special codes. For one thing, the comment can contain **HTML mark-up** commands. HTML is the language that is used to create web pages, and Javadoc comments are meant to be shown on web pages. The `javadoc` tool will copy any HTML commands in the comments to the web pages that it creates. You'll learn some basic HTML in Section 6.2, but as an example, you can add `<p>` to indicate the start of a new paragraph. (Generally, in the absence of HTML commands, blank lines and extra spaces in the comment are ignored.)

In addition to HTML commands, Javadoc comments can include **doc tags**, which are processed as commands by the `javadoc` tool. A doc tag has a name that begins with the character `@`. I will only discuss three tags: `@param`, `@return`, and `@throws`. These tags are used in Javadoc comments for subroutines to provide information about its parameters, its return value, and the exceptions that it might throw. These tags are always placed at the end of the comment, after any description of the subroutine itself. The syntax for using them is:

```
@param  <parameter-name>    <description-of-parameter>

@return  <description-of-return-value>

@throws  <exception-class-name>  <description-of-exception>
```

The `<descriptions>` can extend over several lines. The description ends at the next tag or at the end of the comment. You can include a `@param` tag for every parameter of the subroutine and a `@throws` for as many types of exception as you want to document. You should have a `@return` tag only for a non-void subroutine. These tags do not have to be given in any particular order.

Here is an example that doesn't do anything exciting but that does use all three types of doc tag:

```
/**
 * This subroutine computes the area of a rectangle, given its width
 * and its height. The length and the width should be positive numbers.
 * @param width the length of one side of the rectangle
 * @param height the length the second side of the rectangle
 * @return the area of the rectangle
 * @throws IllegalArgumentException if either the width or the height
 *         is a negative number.
 */
public static double areaOfRectangle( double length, double width ) {
    if ( width < 0 || height < 0 )
        throw new IllegalArgumentException("Sides must have positive length.");
}
```

```
double area;  
area = width * height;  
return area;  
}
```

I will use Javadoc comments for some of my examples. I encourage you to use them in your own code, even if you don't plan to generate Web page documentation of your work, since it's a standard format that other Java programmers will be familiar with.

If you do want to create Web-page documentation, you need to run the `javadoc` tool. This tool is available as a command in the Java Development Kit that was discussed in Section 2.6. You can use `javadoc` in a command line interface similarly to the way that the `javac` and `java` commands are used. Javadoc can also be applied in the Eclipse integrated development environment that was also discussed in Section 2.6: Just right-click the class or package that you want to document in the Package Explorer, select "Export," and select "Javadoc" in the window that pops up. I won't go into any of the details here; see the documentation.

4.6 More on Program Design

UNDERSTANDING HOW PROGRAMS WORK is one thing. Designing a program to perform some particular task is another thing altogether. In Section 3.2, I discussed how pseudocode and stepwise refinement can be used to methodically develop an algorithm. We can now see how subroutines can fit into the process.

Stepwise refinement is inherently a top-down process, but the process does have a "bottom," that is, a point at which you stop refining the pseudocode algorithm and translate what you have directly into proper programming language. In the absence of subroutines, the process would not bottom out until you get down to the level of assignment statements and very primitive input/output operations. But if you have subroutines lying around to perform certain useful tasks, you can stop refining as soon as you've managed to express your algorithm in terms of those tasks.

This allows you to add a bottom-up element to the top-down approach of stepwise refinement. Given a problem, you might start by writing some subroutines that perform tasks relevant to the problem domain. The subroutines become a toolbox of ready-made tools that you can integrate into your algorithm as you develop it. (Alternatively, you might be able to buy or find a software toolbox written by someone else, containing subroutines that you can use in your project as black boxes.)

Subroutines can also be helpful even in a strict top-down approach. As you refine your algorithm, you are free at any point to take any sub-task in the algorithm and make it into a subroutine. Developing that subroutine then becomes a separate problem, which you can work on separately. Your main algorithm will merely call the subroutine. This, of course, is just a way of breaking your problem down into separate, smaller problems. It is still a top-down approach because the top-down analysis of the problem tells you what subroutines to write. In the bottom-up approach, you start by writing or obtaining subroutines that are relevant to the problem domain, and you build your solution to the problem on top of that foundation of subroutines.

4.6.1 Preconditions and Postconditions

When working with subroutines as building blocks, it is important to be clear about how a subroutine interacts with the rest of the program. This interaction is specified by the *contract*

of the subroutine, as discussed in Section 4.1. A convenient way to express the contract of a subroutine is in terms of *preconditions* and *postconditions*.

The precondition of a subroutine is something that must be true when the subroutine is called, if the subroutine is to work correctly. For example, for the built-in function `Math.sqrt(x)`, a precondition is that the parameter, `x`, is greater than or equal to zero, since it is not possible to take the square root of a negative number. In terms of a contract, a precondition represents an obligation of the *caller* of the subroutine. If you call a subroutine without meeting its precondition, then there is no reason to expect it to work properly. The program might crash or give incorrect results, but you can only blame yourself, not the subroutine.

A postcondition of a subroutine represents the other side of the contract. It is something that will be true after the subroutine has run (assuming that its preconditions were met—and that there are no bugs in the subroutine). The postcondition of the function `Math.sqrt()` is that the square of the value that is returned by this function is equal to the parameter that is provided when the subroutine is called. Of course, this will only be true if the precondition—that the parameter is greater than or equal to zero—is met. A postcondition of the built-in subroutine `System.out.print()` is that the value of the parameter has been displayed on the screen.

Preconditions most often give restrictions on the acceptable values of parameters, as in the example of `Math.sqrt(x)`. However, they can also refer to global variables that are used in the subroutine. The postcondition of a subroutine specifies the task that it performs. For a function, the postcondition should specify the value that the function returns.

Subroutines are often described by comments that explicitly specify their preconditions and postconditions. When you are given a pre-written subroutine, a statement of its preconditions and postconditions tells you how to use it and what it does. When you are assigned to write a subroutine, the preconditions and postconditions give you an exact specification of what the subroutine is expected to do. I will use this approach in the example that constitutes the rest of this section. The comments are given in the form of Javadoc comments, but I will explicitly label the preconditions and postconditions. (Many computer scientists think that new doc tags `@precondition` and `@postcondition` should be added to the Javadoc system for explicit labeling of preconditions and postconditions, but that has not yet been done.)

4.6.2 A Design Example

Let's work through an example of program design using subroutines. In this example, we will use prewritten subroutines as building blocks and we will also design new subroutines that we need to complete the project.

Suppose that I have found an already-written class called `Mosaic`. This class allows a program to work with a window that displays little colored rectangles arranged in rows and columns. The window can be opened, closed, and otherwise manipulated with static member subroutines defined in the `Mosaic` class. In fact, the class defines a toolbox or API that can be used for working with such windows. Here are some of the available routines in the API, with Javadoc-style comments:

```
/**
 * Opens a "mosaic" window on the screen.
 *
 * Precondition:  The parameters rows, cols, w, and h are positive integers.
 * Postcondition: A window is open on the screen that can display rows and
 *                columns of colored rectangles. Each rectangle is w pixels
```

```

*           wide and h pixels high. The number of rows is given by
*           the first parameter and the number of columns by the
*           second. Initially, all rectangles are black.
* Note: The rows are numbered from 0 to rows - 1, and the columns are
* numbered from 0 to cols - 1.
*/
public static void open(int rows, int cols, int w, int h)

/**
* Sets the color of one of the rectangles in the window.
*
* Precondition: row and col are in the valid range of row and column numbers,
*               and r, g, and b are in the range 0 to 255, inclusive.
* Postcondition: The color of the rectangle in row number row and column
*               number col has been set to the color specified by r, g,
*               and b. r gives the amount of red in the color with 0
*               representing no red and 255 representing the maximum
*               possible amount of red. The larger the value of r, the
*               more red in the color. g and b work similarly for the
*               green and blue color components.
*/
public static void setColor(int row, int col, int r, int g, int b)

/**
* Gets the red component of the color of one of the rectangles.
*
* Precondition: row and col are in the valid range of row and column numbers.
* Postcondition: The red component of the color of the specified rectangle is
*               returned as an integer in the range 0 to 255 inclusive.
*/
public static int getRed(int row, int col)

/**
* Like getRed, but returns the green component of the color.
*/
public static int getGreen(int row, int col)

/**
* Like getRed, but returns the blue component of the color.
*/
public static int getBlue(int row, int col)

/**
* Tests whether the mosaic window is currently open.
*
* Precondition: None.
* Postcondition: The return value is true if the window is open when this
*               function is called, and it is false if the window is
*               closed.
*/
public static boolean isOpen()

/**

```



```

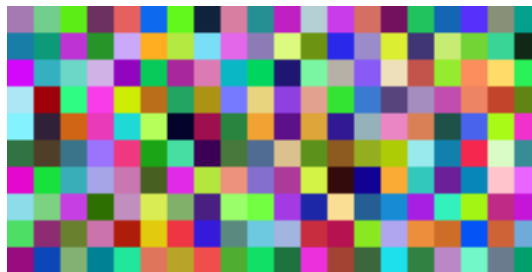
* Inserts a delay in the program (to regulate the speed at which the colors
* are changed, for example).
*
* Precondition:  milliseconds is a positive integer.
* Postcondition: The program has paused for at least the specified number
*                of milliseconds, where one second is equal to 1000
*                milliseconds.
*/
public static void delay(int milliseconds)

```

Remember that these subroutines are members of the `Mosaic` class, so when they are called from outside `Mosaic`, the name of the class must be included as part of the name of the routine. For example, we'll have to use the name `Mosaic.isOpen()` rather than simply `isOpen()`.

* * *

My idea is to use the `Mosaic` class as the basis for a neat animation. I want to fill the window with randomly colored squares, and then randomly change the colors in a loop that continues as long as the window is open. “Randomly change the colors” could mean a lot of different things, but after thinking for a while, I decide it would be interesting to have a “disturbance” that wanders randomly around the window, changing the color of each square that it encounters. Here's a picture showing what the contents of the window might look like at one point in time:



With basic routines for manipulating the window as a foundation, I can turn to the specific problem at hand. A basic outline for my program is

```

Open a Mosaic window
Fill window with random colors;
Move around, changing squares at random.

```

Filling the window with random colors seems like a nice coherent task that I can work on separately, so let's decide to write a separate subroutine to do it. The third step can be expanded a bit more, into the steps: Start in the middle of the window, then keep moving to a new square and changing the color of that square. This should continue as long as the mosaic window is still open. Thus we can refine the algorithm to:

```

Open a Mosaic window
Fill window with random colors;
Set the current position to the middle square in the window;
As long as the mosaic window is open:
    Randomly change color of the square at the current position;
    Move current position up, down, left, or right, at random;

```

I need to represent the current position in some way. That can be done with two **int** variables named `currentRow` and `currentColumn` that hold the row number and the column number of the square where the disturbance is currently located. I'll use 10 rows and 20 columns of squares in my mosaic, so setting the current position to be in the center means setting `currentRow` to 5 and `currentColumn` to 10. I already have a subroutine, `Mosaic.open()`, to open the window, and I have a function, `Mosaic.isOpen()`, to test whether the window is open. To keep the main routine simple, I decide that I will write two more subroutines of my own to carry out the two tasks in the while loop. The algorithm can then be written in Java as:

```
Mosaic.open(10,20,10,10)
fillWithRandomColors();
currentRow = 5;           // Middle row, halfway down the window.
currentColumn = 10;       // Middle column.
while ( Mosaic.isOpen() ) {
    changeToRandomColor(currentRow, currentColumn);
    randomMove();
}
```

With the proper wrapper, this is essentially the `main()` routine of my program. It turns out I have to make one small modification: To prevent the animation from running too fast, the line "`Mosaic.delay(20);`" is added to the `while` loop.

The `main()` routine is taken care of, but to complete the program, I still have to write the subroutines `fillWithRandomColors()`, `changeToRandomColor(int,int)`, and `randomMove()`. Writing each of these subroutines is a separate, small task. The `fillWithRandomColors()` routine is defined by the postcondition that "each of the rectangles in the mosaic has been changed to a random color." Pseudocode for an algorithm to accomplish this task can be given as:

```
For each row:
  For each column:
    set the square in that row and column to a random color
```

"For each row" and "for each column" can be implemented as for loops. We've already planned to write a subroutine `changeToRandomColor` that can be used to set the color. (The possibility of reusing subroutines in several places is one of the big payoffs of using them!) So, `fillWithRandomColors()` can be written in proper Java as:

```
static void fillWithRandomColors() {
    for (int row = 0; row < 10; row++)
        for (int column = 0; column < 20; column++)
            changeToRandomColor(row,column);
}
```

Turning to the `changeToRandomColor` subroutine, we already have a method in the `Mosaic` class, `Mosaic.setColor()`, that can be used to change the color of a square. If we want a random color, we just have to choose random values for **r**, **g**, and **b**. According to the precondition of the `Mosaic.setColor()` subroutine, these random values must be integers in the range from 0 to 255. A formula for randomly selecting such an integer is "`(int)(256*Math.random())`". So the random color subroutine becomes:

```
static void changeToRandomColor(int rowNum, int colNum) {
    int red = (int)(256*Math.random());
    int green = (int)(256*Math.random());
    int blue = (int)(256*Math.random());
```

```

        mosaic.setColor(rowNum,colNum,red,green,blue);
    }

```

Finally, consider the `randomMove` subroutine, which is supposed to randomly move the disturbance up, down, left, or right. To make a random choice among four directions, we can choose a random integer in the range 0 to 3. If the integer is 0, move in one direction; if it is 1, move in another direction; and so on. The position of the disturbance is given by the variables `currentRow` and `currentColumn`. To “move up” means to subtract 1 from `currentRow`. This leaves open the question of what to do if `currentRow` becomes -1, which would put the disturbance above the window. Rather than let this happen, I decide to move the disturbance to the opposite edge of the applet by setting `currentRow` to 9. (Remember that the 10 rows are numbered from 0 to 9.) Moving the disturbance down, left, or right is handled similarly. If we use a `switch` statement to decide which direction to move, the code for `randomMove` becomes:

```

int directionNum;
directionNum = (int)(4*Math.random());
switch (directionNum) {
    case 0: // move up
        currentRow--;
        if (currentRow < 0) // CurrentRow is outside the mosaic;
            currentRow = 9; // move it to the opposite edge.
        break;
    case 1: // move right
        currentColumn++;
        if (currentColumn >= 20)
            currentColumn = 0;
        break;
    case 2: // move down
        currentRow++;
        if (currentRow >= 10)
            currentRow = 0;
        break;
    case 3: // move left
        currentColumn--;
        if (currentColumn < 0)
            currentColumn = 19;
        break;
}

```

4.6.3 The Program

Putting this all together, we get the following complete program. Note that I’ve added Javadoc-style comments for the class itself and for each of the subroutines. The variables `currentRow` and `currentColumn` are defined as static members of the class, rather than local variables, because each of them is used in several different subroutines. This program actually depends on two other classes, `Mosaic` and another class called `MosaicCanvas` that is used by `Mosaic`. If you want to compile and run this program, both of these classes must be available to the program.

```

/**
 * This program opens a window full of randomly colored squares. A "disturbance"
 * moves randomly around in the window, randomly changing the color of each
 * square that it visits. The program runs until the user closes the window.
 */

public class RandomMosaicWalk {

    static int currentRow;    // Row currently containing the disturbance.
    static int currentColumn; // Column currently containing disturbance.

    /**
     * The main program creates the window, fills it with random colors,
     * and then moves the disturbances in a random walk around the window
     * as long as the window is open.
     */
    public static void main(String[] args) {
        Mosaic.open(10,20,10,10);
        fillWithRandomColors();
        currentRow = 5;    // start at center of window
        currentColumn = 10;
        while (Mosaic.isOpen()) {
            changeToRandomColor(currentRow, currentColumn);
            randomMove();
            Mosaic.delay(20);
        }
    } // end main

    /**
     * Fills the window with randomly colored squares.
     * Precondition: The mosaic window is open.
     * Postcondition: Each square has been set to a random color.
     */
    static void fillWithRandomColors() {
        for (int row=0; row < 10; row++) {
            for (int column=0; column < 20; column++) {
                changeToRandomColor(row, column);
            }
        }
    } // end fillWithRandomColors

    /**
     * Changes one square to a new randomly selected color.
     * Precondition: The specified rowNum and colNum are in the valid range
     *                of row and column numbers.
     * Postcondition: The square in the specified row and column has
     *                been set to a random color.
     * @param rowNum the row number of the square, counting rows down
     *                from 0 at the top
     * @param colNum the column number of the square, counting columns over
     *                from 0 at the left
     */
    static void changeToRandomColor(int rowNum, int colNum) {
        int red = (int)(256*Math.random());    // Choose random levels in range
        int green = (int)(256*Math.random());  //      0 to 255 for red, green,
        int blue = (int)(256*Math.random());   //      and blue color components.
    }
}

```

```

        Mosaic.setColor(rowNum,colNum,red,green,blue);
    } // end of changeToRandomColor()

/**
 * Move the disturbance.
 * Precondition:  The global variables currentRow and currentColumn
 *               are within the legal range of row and column numbers.
 * Postcondition: currentRow or currentColumn is changed to one of the
 *               neighboring positions in the grid -- up, down, left, or
 *               right from the current position.  If this moves the
 *               position outside of the grid, then it is moved to the
 *               opposite edge of the grid.
 */
static void randomMove() {
    int directionNum; // Randomly set to 0, 1, 2, or 3 to choose direction.
    directionNum = (int)(4*Math.random());
    switch (directionNum) {
        case 0: // move up
            currentRow--;
            if (currentRow < 0)
                currentRow = 9;
            break;
        case 1: // move right
            currentColumn++;
            if (currentColumn >= 20)
                currentColumn = 0;
            break;
        case 2: // move down
            currentRow++;
            if (currentRow >= 10)
                currentRow = 0;
            break;
        case 3: // move left
            currentColumn--;
            if (currentColumn < 0)
                currentColumn = 19;
            break;
    }
} // end randomMove

} // end class RandomMosaicWalk

```

4.7 The Truth About Declarations

NAMES ARE FUNDAMENTAL TO PROGRAMMING, as I said a few chapters ago. There are a lot of details involved in declaring and using names. I have been avoiding some of those details. In this section, I'll reveal most of the truth (although still not the full truth) about declaring and using variables in Java. The material in the subsections "Initialization in Declarations" and "Named Constants" is particularly important, since I will be using it regularly in future chapters.

4.7.1 Initialization in Declarations

When a variable declaration is executed, memory is allocated for the variable. This memory must be initialized to contain some definite value before the variable can be used in an expression. In the case of a local variable, the declaration is often followed closely by an assignment statement that does the initialization. For example,

```
int count;    // Declare a variable named count.
count = 0;    // Give count its initial value.
```

However, the truth about declaration statements is that it is legal to include the initialization of the variable in the declaration statement. The two statements above can therefore be abbreviated as

```
int count = 0; // Declare count and give it an initial value.
```

The computer still executes this statement in two steps: Declare the variable `count`, then assign the value 0 to the newly created variable. The initial value does not have to be a constant. It can be any expression. It is legal to initialize several variables in one declaration statement. For example,

```
char firstInitial = 'D', secondInitial = 'E';

int x, y = 1;    // OK, but only y has been initialized!

int N = 3, M = N+2; // OK, N is initialized
                  //           before its value is used.
```

This feature is especially common in `for` loops, since it makes it possible to declare a loop control variable at the same point in the loop where it is initialized. Since the loop control variable generally has nothing to do with the rest of the program outside the loop, it's reasonable to have its declaration in the part of the program where it's actually used. For example:

```
for ( int i = 0; i < 10; i++ ) {
    System.out.println(i);
}
```

Again, you should remember that this is simply an abbreviation for the following, where I've added an extra pair of braces to show that `i` is considered to be local to the `for` statement and no longer exists after the `for` loop ends:

```
{
    int i;
    for ( i = 0; i < 10; i++ ) {
        System.out.println(i);
    }
}
```

(You might recall, by the way, that for “for-each” loops, the special type of `for` statement that is used with enumerated types, declaring the variable in the `for` is **required**. See Subsection 3.4.4.)

A member variable can also be initialized at the point where it is declared, just as for a local variable. For example:

```
public class Bank {
    static double interestRate = 0.05;
    static int maxWithdrawal = 200;
```

```

        .
        . // More variables and subroutines.
        .
    }

```

A static member variable is created as soon as the class is loaded by the Java interpreter, and the initialization is also done at that time. In the case of member variables, this is not simply an abbreviation for a declaration followed by an assignment statement. Declaration statements are the only type of statement that can occur outside of a subroutine. Assignment statements cannot, so the following is illegal:

```

public class Bank {
    static double interestRate;
    interestRate = 0.05; // ILLEGAL:
    .                  // Can't be outside a subroutine!
    .
    .
}

```

Because of this, declarations of member variables often include initial values. In fact, as mentioned in Subsection 4.2.4, if no initial value is provided for a member variable, then a default initial value is used. For example, when declaring an integer member variable, `count`, “`static int count;`” is equivalent to “`static int count = 0;`”.

4.7.2 Named Constants

Sometimes, the value of a variable is not supposed to change after it is initialized. For example, in the above example where `interestRate` is initialized to the value 0.05, it's quite possible that that is meant to be the value throughout the entire program. In this case, the programmer is probably defining the variable, `interestRate`, to give a meaningful name to the otherwise meaningless number, 0.05. It's easier to understand what's going on when a program says “`principal += principal*interestRate;`” rather than “`principal += principal*0.05;`”.

In Java, the modifier “`final`” can be applied to a variable declaration to ensure that the value stored in the variable cannot be changed after the variable has been initialized. For example, if the member variable `interestRate` is declared with

```
final static double interestRate = 0.05;
```

then it would be impossible for the value of `interestRate` to change anywhere else in the program. Any assignment statement that tries to assign a value to `interestRate` will be rejected by the computer as a syntax error when the program is compiled.

It is legal to apply the `final` modifier to local variables and even to formal parameters, but it is most useful for member variables. I will often refer to a static member variable that is declared to be `final` as a *named constant*, since its value remains constant for the whole time that the program is running. The readability of a program can be greatly enhanced by using named constants to give meaningful names to important quantities in the program. A recommended style rule for named constants is to give them names that consist entirely of upper case letters, with underscore characters to separate words if necessary. For example, the preferred style for the interest rate constant would be

```
final static double INTEREST_RATE = 0.05;
```

This is the style that is generally used in Java's standard classes, which define many named constants. For example, we have already seen that the *Math* class contains a variable `Math.PI`. This variable is declared in the *Math* class as a “public final static” variable of type **double**.

Similarly, the `Color` class contains named constants such as `Color.RED` and `Color.YELLOW` which are public final static variables of type `Color`. Many named constants are created just to give meaningful names to be used as parameters in subroutine calls. For example, the standard class named `Font` contains named constants `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. These constants are used for specifying different styles of text when calling various subroutines in the `Font` class.

Enumerated type constants (See Subsection 2.3.3.) are also examples of named constants. The enumerated type definition

```
enum Alignment { LEFT, RIGHT, CENTER }
```

defines the constants `Alignment.LEFT`, `Alignment.RIGHT`, and `Alignment.CENTER`. Technically, *Alignment* is a class, and the three constants are public final static members of that class. Defining the enumerated type is similar to defining three constants of type, say, `int`:

```
public static final int ALIGNMENT_LEFT = 0;
public static final int ALIGNMENT_RIGHT = 1;
public static final int ALIGNMENT_CENTER = 2;
```

In fact, this is how things were generally done before the introduction of enumerated types in Java 5.0, and it is what is done with the constants `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC` mentioned above. Using the integer constants, you could define a variable of type `int` and assign it the values `ALIGNMENT_LEFT`, `ALIGNMENT_RIGHT`, or `ALIGNMENT_CENTER` to represent different types of alignment. The only problem with this is that the computer has no way of knowing that you intend the value of the variable to represent an alignment, and it will not raise any objection if the value that is assigned to the variable is not one of the three valid alignment values.

With the enumerated type, on the other hand, the only values that can be assigned to a variable of type *Alignment* are the constant values that are listed in the definition of the enumerated type. Any attempt to assign an invalid value to the variable is a syntax error which the computer will detect when the program is compiled. This extra safety is one of the major advantages of enumerated types.

* * *

Curiously enough, one of the major reasons to use named constants is that it's easy to change the value of a named constant. Of course, the value can't change while the program is running. But between runs of the program, it's easy to change the value in the source code and recompile the program. Consider the interest rate example. It's quite possible that the value of the interest rate is used many times throughout the program. Suppose that the bank changes the interest rate and the program has to be modified. If the literal number 0.05 were used throughout the program, the programmer would have to track down each place where the interest rate is used in the program and change the rate to the new value. (This is made even harder by the fact that the number 0.05 might occur in the program with other meanings besides the interest rate, as well as by the fact that someone might have used 0.025 to represent half the interest rate.) On the other hand, if the named constant `INTEREST_RATE` is declared and used consistently throughout the program, then only the single line where the constant is initialized needs to be changed.

As an extended example, I will give a new version of the `RandomMosaicWalk` program from the previous section. This version uses named constants to represent the number of rows in the mosaic, the number of columns, and the size of each little square. The three constants are declared as `final static` member variables with the lines:


```
final static int ROWS = 30;           // Number of rows in mosaic.
final static int COLUMNS = 30;      // Number of columns in mosaic.
final static int SQUARE_SIZE = 15;  // Size of each square in mosaic.
```

The rest of the program is carefully modified to use the named constants. For example, in the new version of the program, the Mosaic window is opened with the statement

```
Mosaic.open(ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE);
```

Sometimes, it's not easy to find all the places where a named constant needs to be used. If you don't use the named constant consistently, you've more or less defeated the purpose. It's always a good idea to run a program using several different values for any named constants, to test that it works properly in all cases.

Here is the complete new program, `RandomMosaicWalk2`, with all modifications from the previous version shown in *italic*. I've left out some of the comments to save space.

```
public class RandomMosaicWalk2 {

    final static int ROWS = 30;           // Number of rows in mosaic.
    final static int COLUMNS = 30;      // Number of columns in mosaic.
    final static int SQUARE_SIZE = 15;  // Size of each square in mosaic.

    static int currentRow;    // Row currently containing the disturbance.
    static int currentColumn; // Column currently containing disturbance.

    public static void main(String[] args) {
        Mosaic.open( ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE );
        fillWithRandomColors();
        currentRow = ROWS / 2;    // start at center of window
        currentColumn = COLUMNS / 2;
        while (Mosaic.isOpen()) {
            changeToRandomColor(currentRow, currentColumn);
            randomMove();
            Mosaic.delay(20);
        }
    } // end main

    static void fillWithRandomColors() {
        for (int row=0; row < ROWS; row++) {
            for (int column=0; column < COLUMNS; column++) {
                changeToRandomColor(row, column);
            }
        }
    } // end fillWithRandomColors

    static void changeToRandomColor(int rowNum, int colNum) {
        int red = (int)(256*Math.random());    // Choose random levels in range
        int green = (int)(256*Math.random()); //      0 to 255 for red, green,
        int blue = (int)(256*Math.random());  //      and blue color components.
        Mosaic.setColor(rowNum,colNum,red,green,blue);
    } // end changeToRandomColor

    static void randomMove() {
        int directionNum; // Randomly set to 0, 1, 2, or 3 to choose direction.
        directionNum = (int)(4*Math.random());
        switch (directionNum) {
            case 0: // move up
```

```

        currentRow--;
        if (currentRow < 0)
            currentRow = ROWS - 1;
        break;
    case 1: // move right
        currentColumn++;
        if (currentColumn >= COLUMNS)
            currentColumn = 0;
        break;
    case 2: // move down
        currentRow ++;
        if (currentRow >= ROWS)
            currentRow = 0;
        break;
    case 3: // move left
        currentColumn--;
        if (currentColumn < 0)
            currentColumn = COLUMNS - 1;
        break;
    }
} // end randomMove
} // end class RandomMosaicWalk2

```

4.7.3 Naming and Scope Rules

When a variable declaration is executed, memory is allocated for that variable. The variable name can be used in at least some part of the program source code to refer to that memory or to the data that is stored in the memory. The portion of the program source code where the variable name is valid is called the *scope* of the variable. Similarly, we can refer to the scope of subroutine names and formal parameter names.

For static member subroutines, scope is straightforward. The scope of a static subroutine is the entire source code of the class in which it is defined. That is, it is possible to call the subroutine from any point in the class, including at a point in the source code before the point where the definition of the subroutine appears. It is even possible to call a subroutine from within itself. This is an example of something called “recursion,” a fairly advanced topic that we will return to later.

For a variable that is declared as a static member variable in a class, the situation is similar, but with one complication. It is legal to have a local variable or a formal parameter that has the same name as a member variable. In that case, within the scope of the local variable or parameter, the member variable is *hidden*. Consider, for example, a class named `Game` that has the form:

```

public class Game {
    static int count; // member variable

    static void playGame() {
        int count; // local variable
        .
        . // Some statements to define playGame()
        .
    }
}

```

```

    .
    .    // More variables and subroutines.
    .
}    // end Game

```

In the statements that make up the body of the `playGame()` subroutine, the name “`count`” refers to the local variable. In the rest of the `Game` class, “`count`” refers to the member variable, unless hidden by other local variables or parameters named `count`. However, there is one further complication. The member variable named `count` can also be referred to by the full name `Game.count`. Usually, the full name is only used outside the class where `count` is defined. However, there is no rule against using it inside the class. The full name, `Game.count`, can be used inside the `playGame()` subroutine to refer to the member variable. So, the full scope rule is that the scope of a static member variable includes the entire class in which it is defined, but where the simple name of the member variable is hidden by a local variable or formal parameter name, the member variable must be referred to by its full name of the form $\langle \text{className} \rangle . \langle \text{variableName} \rangle$. (Scope rules for non-static members are similar to those for static members, except that, as we shall see, non-static members cannot be used in static subroutines.)

The scope of a formal parameter of a subroutine is the block that makes up the body of the subroutine. The scope of a local variable extends from the declaration statement that defines the variable to the end of the block in which the declaration occurs. As noted above, it is possible to declare a loop control variable of a `for` loop in the `for` statement, as in “`for (int i=0; i < 10; i++)`”. The scope of such a declaration is considered as a special case: It is valid only within the `for` statement and does not extend to the remainder of the block that contains the `for` statement.

It is not legal to redefine the name of a formal parameter or local variable within its scope, even in a nested block. For example, this is not allowed:

```

void badSub(int y) {
    int x;
    while (y > 0) {
        int x;    // ERROR: x is already defined.
        .
        .
        .
    }
}

```

In many languages, this would be legal; the declaration of `x` in the `while` loop would hide the original declaration. It is not legal in Java; however, once the block in which a variable is declared ends, its name does become available for reuse in Java. For example:

```

void goodSub(int y) {
    while (y > 10) {
        int x;
        .
        .
        .
        // The scope of x ends here.
    }
    while (y > 0) {

```

```
        int x;  // OK: Previous declaration of x has expired.
        .
        .
        .
    }
}
```

You might wonder whether local variable names can hide subroutine names. This can't happen, for a reason that might be surprising. There is no rule that variables and subroutines have to have different names. The computer can always tell whether a name refers to a variable or to a subroutine, because a subroutine name is always followed by a left parenthesis. It's perfectly legal to have a variable called `count` and a subroutine called `count` in the same class. (This is one reason why I often write subroutine names with parentheses, as when I talk about the `main()` routine. It's a good idea to think of the parentheses as part of the name.) Even more is true: It's legal to reuse class names to name variables and subroutines. The syntax rules of Java guarantee that the computer can always tell when a name is being used as a class name. A class name is a type, and so it can be used to declare variables and formal parameters and to specify the return type of a function. This means that you could legally have a class called `Insanity` in which you declare a function

```
static Insanity Insanity( Insanity Insanity ) { ... }
```

The first `Insanity` is the return type of the function. The second is the function name, the third is the type of the formal parameter, and the fourth is a formal parameter name. However, please remember that not everything that is possible is a good idea!

Exercises for Chapter 4

1. To “capitalize” a string means to change the first letter of each word in the string to upper case (if it is not already upper case). For example, a capitalized version of “Now is the time to act!” is “Now Is The Time To Act!”. Write a subroutine named `printCapitalized` that will print a capitalized version of a string to standard output. The string to be printed should be a parameter to the subroutine. Test your subroutine with a `main()` routine that gets a line of input from the user and applies the subroutine to it.

Note that a letter is the first letter of a word if it is not immediately preceded in the string by another letter. Recall that there is a standard **boolean**-valued function `Character.isLetter(char)` that can be used to test whether its parameter is a letter. There is another standard **char**-valued function, `Character.toUpperCase(char)`, that returns a capitalized version of the single character passed to it as a parameter. That is, if the parameter is a letter, it returns the upper-case version. If the parameter is not a letter, it just returns a copy of the parameter.

2. The hexadecimal digits are the ordinary, base-10 digits '0' through '9' plus the letters 'A' through 'F'. In the hexadecimal system, these digits represent the values 0 through 15, respectively. Write a function named `hexValue` that uses a `switch` statement to find the hexadecimal value of a given character. The character is a parameter to the function, and its hexadecimal value is the return value of the function. You should count lower case letters 'a' through 'f' as having the same value as the corresponding upper case letters. If the parameter is not one of the legal hexadecimal digits, return -1 as the value of the function.

A hexadecimal integer is a sequence of hexadecimal digits, such as 34A7, FF8, 174204, or FADE. If `str` is a string containing a hexadecimal integer, then the corresponding base-10 integer can be computed as follows:

```
value = 0;
for ( i = 0; i < str.length(); i++ )
    value = value*16 + hexValue( str.charAt(i) );
```

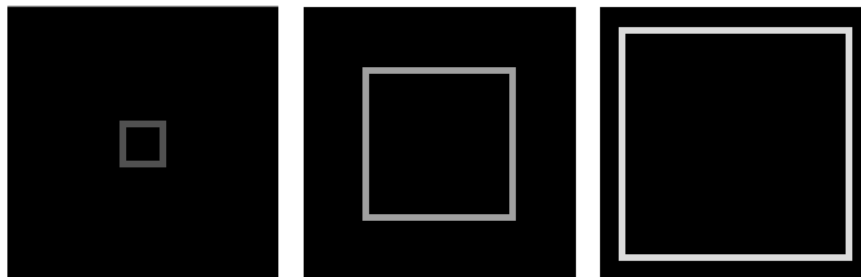
Of course, this is not valid if `str` contains any characters that are not hexadecimal digits. Write a program that reads a string from the user. If all the characters in the string are hexadecimal digits, print out the corresponding base-10 value. If not, print out an error message.

3. Write a function that simulates rolling a pair of dice until the total on the dice comes up to be a given number. The number that you are rolling for is a parameter to the function. The number of times you have to roll the dice is the return value of the function. The parameter should be one of the possible totals: 2, 3, ..., 12. The function should throw an *IllegalArgumentException* if this is not the case. Use your function in a program that computes and prints the number of rolls it takes to get snake eyes. (Snake eyes means that the total showing on the dice is 2.)
4. This exercise builds on Exercise 4.3. Every time you roll the dice repeatedly, trying to get a given total, the number of rolls it takes can be different. The question naturally arises, what's the average number of rolls to get a given total? Write a function that performs the experiment of rolling to get a given total 10000 times. The desired total is

a parameter to the subroutine. The average number of rolls is the return value. Each individual experiment should be done by calling the function you wrote for Exercise 4.3. Now, write a main program that will call your function once for each of the possible totals (2, 3, ..., 12). It should make a table of the results, something like:

Total On Dice	Average Number of Rolls
-----	-----
2	35.8382
3	18.0607
.	.
.	.

- The sample program *RandomMosaicWalk.java* from Section 4.6 shows a “disturbance” that wanders around a grid of colored squares. When the disturbance visits a square, the color of that square is changed. The applet at the bottom of Section 4.7 in the on-line version of this book shows a variation on this idea. In this applet, all the squares start out with the default color, black. Every time the disturbance visits a square, a small amount is added to the red component of the color of that square. Write a subroutine that will add 25 to the red component of one of the squares in the mosaic. The row and column numbers of the square should be passed as parameters to the subroutine. Recall that you can discover the current red component of the square in row *r* and column *c* with the function call `Mosaic.getRed(r,c)`. Use your subroutine as a substitute for the `changeToRandomColor()` subroutine in the program *RandomMosaicWalk2.java*. (This is the improved version of the program from Section 4.7 that uses named constants for the number of rows, number of columns, and square size.) Set the number of rows and the number of columns to 80. Set the square size to 5.
- For this exercise, you will write another program based on the non-standard `Mosaic` class that was presented in Section 4.6. While the program does not do anything particularly interesting, it’s interesting as a programming problem. An applet that does the same thing as the program can be seen in the on-line version of this book. Here is a picture showing what it looks like at several different times:



The program will show a rectangle that grows from the center of the applet to the edges, getting brighter as it grows. The rectangle is made up of the little squares of the mosaic. You should first write a subroutine that draws a rectangle on a `Mosaic` window. More specifically, write a subroutine named `rectangle` such that the subroutine call statement

```
rectangle(top,left,height,width,r,g,b);
```

will call `Mosaic.setColor(row,col,r,g,b)` for each little square that lies on the outline of a rectangle. The topmost row of the rectangle is specified by `top`. The number of rows in the rectangle is specified by `height` (so the bottommost row is `top+height-1`). The leftmost column of the rectangle is specified by `left`. The number of columns in the rectangle is specified by `width` (so the rightmost column is `left+width-1`.)

The animation loops through the same sequence of steps over and over. In each step, a rectangle is drawn in gray (that is, with all three color components having the same value). There is a pause of 200 milliseconds so the user can see the rectangle. Then the very same rectangle is drawn in black, effectively erasing the gray rectangle. Finally, the variables giving the top row, left column, size, and color level of the rectangle are adjusted to get ready for the next step. In the applet, the color level starts at 50 and increases by 10 after each step. When the rectangle gets to the outer edge of the applet, the loop ends. The animation then starts again at the beginning of the loop. You might want to make a subroutine that does one loop through all the steps of the animation.

The `main()` routine simply opens a Mosaic window and then does the animation loop over and over until the user closes the window. There is a 1000 millisecond delay between one animation loop and the next. Use a Mosaic window that has 41 rows and 41 columns. (I advise you **not** to use named constants for the numbers of rows and columns, since the problem is complicated enough already.)

Quiz on Chapter 4

1. A “black box” has an interface and an implementation. Explain what is meant by the terms *interface* and *implementation*.
2. A subroutine is said to have a *contract*. What is meant by the contract of a subroutine? When you want to use a subroutine, why is it important to understand its contract? The contract has both “syntactic” and “semantic” aspects. What is the syntactic aspect? What is the semantic aspect?
3. Briefly explain how subroutines can be a useful tool in the top-down design of programs.
4. Discuss the concept of *parameters*. What are parameters for? What is the difference between *formal parameters* and *actual parameters*?
5. Give two different reasons for using named constants (declared with the `final` modifier).
6. What is an API? Give an example.
7. Write a subroutine named “stars” that will output a line of stars to standard output. (A star is the character “*”). The number of stars should be given as a parameter to the subroutine. Use a *for* loop. For example, the command “stars(20)” would output

```
*****
```

8. Write a `main()` routine that uses the subroutine that you wrote for Question 7 to output 10 lines of stars with 1 star in the first line, 2 stars in the second line, and so on, as shown below.

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

9. Write a function named `countChars` that has a *String* and a **char** as parameters. The function should count the number of times the character occurs in the string, and it should return the result as the value of the function.
10. Write a subroutine with three parameters of type *int*. The subroutine should determine which of its parameters is smallest. The value of the smallest parameter should be returned as the value of the subroutine.

Chapter 5

Programming in the Large II: Objects and Classes

WHEREAS A SUBROUTINE represents a single task, an object can encapsulate both data (in the form of instance variables) and a number of different tasks or “behaviors” related to that data (in the form of instance methods). Therefore objects provide another, more sophisticated type of structure that can be used to help manage the complexity of large programs.

This chapter covers the creation and use of objects in Java. Section 5.5 covers the central ideas of object-oriented programming: inheritance and polymorphism. However, in this text-book, we will generally use these ideas in a limited form, by creating independent classes and building on existing classes rather than by designing entire hierarchies of classes from scratch. Section 5.6 and Section 5.7 cover some of the many details of object oriented programming in Java. Although these details are used occasionally later in the book, you might want to skim through them now and return to them later when they are actually needed.

5.1 Objects, Instance Methods, and Instance Variables

OBJECT-ORIENTED PROGRAMMING (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem. Programming then consists of finding a sequence of instructions that will accomplish that task. But at the heart of object-oriented programming, instead of tasks we find objects—entities that have behaviors, that hold information, and that can interact with one another. Programming consists of designing a set of objects that somehow model the problem at hand. Software objects in the program can represent real or abstract entities in the problem domain. This is supposed to make the design of the program more natural and hence easier to get right and easier to understand.

To some extent, OOP is just a change in point of view. We can think of an object in standard programming terms as nothing more than a set of variables together with some subroutines for manipulating those variables. In fact, it is possible to use object-oriented techniques in any programming language. However, there is a big difference between a language that makes OOP possible and one that actively supports it. An object-oriented programming language such as Java includes a number of features that make it very different from a standard language. In order to make effective use of those features, you have to “orient” your thinking correctly.

5.1.1 Objects, Classes, and Instances

Objects are closely related to classes. We have already been working with classes for several chapters, and we have seen that a class can contain variables and subroutines. If an object is also a collection of variables and subroutines, how do they differ from classes? And why does it require a different type of thinking to understand and use them effectively? In the one section where we worked with objects rather than classes, Section 3.8, it didn't seem to make much difference: We just left the word “**static**” out of the subroutine definitions!

I have said that classes “describe” objects, or more exactly that the non-static portions of classes describe objects. But it's probably not very clear what this means. The more usual terminology is to say that objects *belong to* classes, but this might not be much clearer. (There is a real shortage of English words to properly distinguish all the concepts involved. An object certainly doesn't “belong” to a class in the same way that a member variable “belongs” to a class.) From the point of view of programming, it is more exact to say that classes are used to create objects. A class is a kind of factory for constructing objects. The non-static parts of the class specify, or describe, what variables and subroutines the objects will contain. This is part of the explanation of how objects differ from classes: Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

Consider a simple class whose job is to group together a few static member variables. For example, the following class could be used to store information about the person who is using the program:

```
class UserData {
    static String name;
    static int age;
}
```

In a program that uses this class, there is only one copy of each of the variables `UserData.name` and `UserData.age`. There can only be one “user,” since we only have memory space to store data about one user. The class, `UserData`, and the variables it contains exist as long as the program runs. Now, consider a similar class that includes non-static variables:

```
class PlayerData {
    String name;
    int age;
}
```

In this case, there is no such variable as `PlayerData.name` or `PlayerData.age`, since `name` and `age` are not static members of `PlayerData`. So, there is nothing much in the class at all—except the potential to create objects. But, it's a lot of potential, since it can be used to create any number of objects! Each object will have its **own** variables called `name` and `age`. There can be many “players” because we can make new objects to represent new players on demand. A program might use this class to store information about multiple players in a game. Each player has a name and an age. When a player joins the game, a new `PlayerData` object can be created to represent that player. If a player leaves the game, the `PlayerData` object that represents that player can be destroyed. A system of objects in the program is being used to dynamically model what is happening in the game. You can't do this with “static” variables!

In Section 3.8, we worked with applets, which are objects. The reason they didn't seem to be any different from classes is because we were only working with one applet in each class that we looked at. But one class can be used to make many applets. Think of an applet that scrolls

a message across a Web page. There could be several such applets on the same page, all created from the same class. If the scrolling message in the applet is stored in a non-static variable, then each applet will have its own variable, and each applet can show a different message. The situation is even clearer if you think about windows, which, like applets, are objects. As a program runs, many windows might be opened and closed, but all those windows can belong to the same class. Here again, we have a dynamic situation where multiple objects are created and destroyed as a program runs.

* * *

An object that belongs to a class is said to be an *instance* of that class. The variables that the object contains are called *instance variables*. The subroutines that the object contains are called *instance methods*. (Recall that in the context of object-oriented programming, *method* is a synonym for “subroutine”. From now on, since we are doing object-oriented programming, I will prefer the term “method.”) For example, if the `PlayerData` class, as defined above, is used to create an object, then that object is an instance of the `PlayerData` class, and `name` and `age` are instance variables in the object. It is important to remember that the class of an object determines the **types** of the instance variables; however, the actual data is contained inside the individual objects, not the class. Thus, each object has its own set of data.

An applet that scrolls a message across a Web page might include a subroutine named `scroll()`. Since the applet is an object, this subroutine is an instance method of the applet. The source code for the method is in the class that is used to create the applet. Still, it’s better to think of the instance method as belonging to the object, not to the class. The non-static subroutines in the class merely specify the instance methods that every object created from the class will contain. The `scroll()` methods in two different applets do the same thing in the sense that they both scroll messages across the screen. But there is a real difference between the two `scroll()` methods. The messages that they scroll can be different. You might say that the method definition in the class specifies what type of behavior the objects will have, but the specific behavior can vary from object to object, depending on the values of their instance variables.

As you can see, the static and the non-static portions of a class are very different things and serve very different purposes. Many classes contain only static members, or only non-static. However, it is possible to mix static and non-static members in a single class, and we’ll see a few examples later in this chapter where it is reasonable to do so. You should distinguish between the **source code** for the class, and the **class itself**. The source code determines both the class and the objects that are created from that class. The “static” definitions in the source code specify the things that are part of the class itself, whereas the non-static definitions in the source code specify things that will become part of every instance object that is created from the class. By the way, static member variables and static member subroutines in a class are sometimes called *class variables* and *class methods*, since they belong to the class itself, rather than to instances of that class.

5.1.2 Fundamentals of Objects

So far, I’ve been talking mostly in generalities, and I haven’t given you much idea what you have to put in a program if you want to work with objects. Let’s look at a specific example to see how it works. Consider this extremely simplified version of a `Student` class, which could be used to store information about students taking a course:

```

public class Student {

    public String name; // Student's name.
    public double test1, test2, test3; // Grades on three tests.

    public double getAverage() { // compute average test grade
        return (test1 + test2 + test3) / 3;
    }

} // end of class Student

```

None of the members of this class are declared to be **static**, so the class exists only for creating objects. This class definition says that any object that is an instance of the **Student** class will include instance variables named **name**, **test1**, **test2**, and **test3**, and it will include an instance method named **getAverage()**. The names and tests in different objects will generally have different values. When called for a particular student, the method **getAverage()** will compute an average using **that student's** test grades. Different students can have different averages. (Again, this is what it means to say that an instance method belongs to an individual object, not to the class.)

In Java, a class is a **type**, similar to the built-in types such as **int** and **boolean**. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a function. For example, a program could define a variable named **std** of type **Student** with the statement

```
Student std;
```

However, declaring a variable does **not** create an object! This is an important point, which is related to this Very Important Fact:

**In Java, no variable can ever hold an object.
A variable can only hold a reference to an object.**

You should think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the **heap** where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a **reference** or **pointer** to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object.

In a program, objects are created using an operator called **new**, which creates an object and returns a reference to that object. For example, assuming that **std** is a variable of type **Student**, declared as above, the assignment statement

```
std = new Student();
```

would create a new object which is an instance of the class **Student**, and it would store a reference to that object in the variable **std**. The value of the variable is a reference to the object, not the object itself. It is not quite true, then, to say that the object is the "value of the variable **std**" (though sometimes it is hard to avoid using this terminology). It is certainly **not at all true** to say that the object is "stored in the variable **std**." The proper terminology is that "the variable **std** *refers to* the object," and I will try to stick to that terminology as much as possible.

So, suppose that the variable **std** refers to an object belonging to the class **Student**. That object has instance variables **name**, **test1**, **test2**, and **test3**. These instance variables can

be referred to as `std.name`, `std.test1`, `std.test2`, and `std.test3`. This follows the usual naming convention that when B is part of A, then the full name of B is A.B. For example, a program might include the lines

```
System.out.println("Hello, " + std.name + ". Your test grades are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
```

This would output the name and test grades from the object to which `std` refers. Similarly, `std` can be used to call the `getAverage()` instance method in the object by saying `std.getAverage()`. To print out the student's average, you could say:

```
System.out.println( "Your average is " + std.getAverage() );
```

More generally, you could use `std.name` any place where a variable of type *String* is legal. You can use it in expressions. You can assign a value to it. You can even use it to call subroutines from the *String* class. For example, `std.name.length()` is the number of characters in the student's name.

It is possible for a variable like `std`, whose type is given by a class, to refer to no object at all. We say in this case that `std` holds a *null reference*. The null reference is written in Java as `"null"`. You can store a null reference in the variable `std` by saying

```
std = null;
```

and you could test whether the value of `std` is null by testing

```
if (std == null) . . .
```

If the value of a variable is `null`, then it is, of course, illegal to refer to instance variables or instance methods through that variable—since there is no object, and hence no instance variables to refer to. For example, if the value of the variable `std` is `null`, then it would be illegal to refer to `std.test1`. If your program attempts to use a null reference illegally like this, the result is an error called a *null pointer exception*.

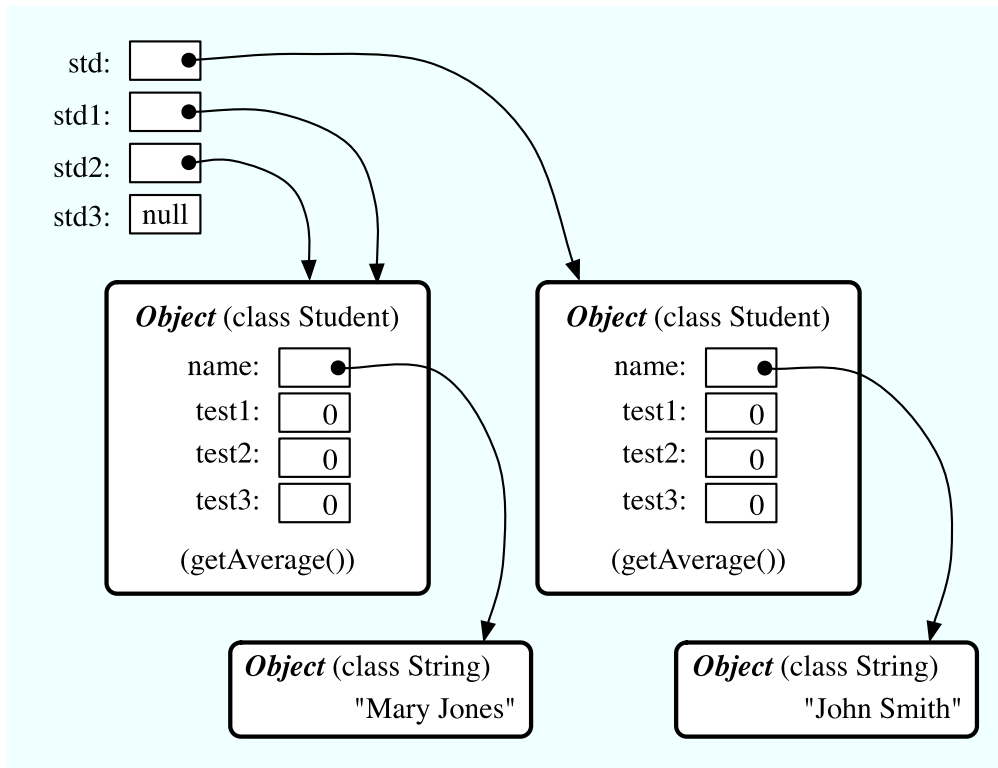
Let's look at a sequence of statements that work with objects:

```
Student std, std1,      // Declare four variables of
    std2, std3;         //   type Student.
std = new Student();    // Create a new object belonging
                        //   to the class Student, and
                        //   store a reference to that
                        //   object in the variable std.
std1 = new Student();   // Create a second Student object
                        //   and store a reference to
                        //   it in the variable std1.
std2 = std1;            // Copy the reference value in std1
                        //   into the variable std2.
std3 = null;            // Store a null reference in the
                        //   variable std3.

std.name = "John Smith"; // Set values of some instance variables.
std1.name = "Mary Jones";

// (Other instance variables have default
//   initial values of zero.)
```

After the computer executes these statements, the situation in the computer's memory looks like this:



This picture shows variables as little boxes, labeled with the names of the variables. Objects are shown as boxes with round corners. When a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the object. The variable `std3`, with a value of `null`, doesn't point anywhere. The arrows from `std1` and `std2` both point to the same object. This illustrates a Very Important Point:

**When one object variable is assigned
to another, only a reference is copied.
The object referred to is not copied.**

When the assignment “`std2 = std1;`” was executed, no new object was created. Instead, `std2` was set to refer to the very same object that `std1` refers to. This has some consequences that might be surprising. For example, `std1.name` and `std2.name` are two different names for the same variable, namely the instance variable in the object that both `std1` and `std2` refer to. After the string “Mary Jones” is assigned to the variable `std1.name`, it is also true that the value of `std2.name` is “Mary Jones”. There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, “The object is not in the variable. The variable just holds a pointer to the object.”

You can test objects for equality and inequality using the operators `==` and `!=`, but here again, the semantics are different from what you are used to. When you make a test “`if (std1 == std2)`”, you are testing whether the values stored in `std1` and `std2` are the same. But the values are references to objects, not objects. So, you are testing whether `std1` and `std2` refer to the same object, that is, whether they point to the same location

in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether `std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3 == std2.test3 && std1.name.equals(std2.name)`".

I've remarked previously that `Strings` are objects, and I've shown the strings "Mary Jones" and "John Smith" as objects in the above illustration. A variable of type `String` can only hold a reference to a string, not the string itself. It could also hold the value `null`, meaning that it does not refer to any string at all. This explains why using the `==` operator to test strings for equality is not a good idea. Suppose that `greeting` is a variable of type `String`, and that the string it refers to is "Hello". Then would the test `greeting == "Hello"` be true? Well, maybe, maybe not. The variable `greeting` and the `String` literal "Hello" each refer to a string that contains the characters H-e-l-l-o. But the strings could still be different objects, that just happen to contain the same characters. The function `greeting.equals("Hello")` tests whether `greeting` and "Hello" contain the same characters, which is almost certainly the question you want to ask. The expression `greeting == "Hello"` tests whether `greeting` and "Hello" contain the same characters **stored in the same memory location**.

* * *

The fact that variables hold references to objects, not objects themselves, has a couple of other consequences that you should be aware of. They follow logically, if you just keep in mind the basic fact that the object is not stored in the variable. The object is somewhere else; the variable points to it.

Suppose that a variable that refers to an object is declared to be `final`. This means that the value stored in the variable can never be changed, once the variable has been initialized. The value stored in the variable is a reference to the object. So the variable will continue to refer to the same object as long as the variable exists. However, this does not prevent the data **in the object** from changing. The variable is `final`, not the object. It's perfectly legal to say

```
final Student stu = new Student();

stu.name = "John Doe"; // Change data in the object;
                       // The value stored in stu is not changed!
                       // It still refers to the same object.
```

Next, suppose that `obj` is a variable that refers to an object. Let's consider what happens when `obj` is passed as an actual parameter to a subroutine. The value of `obj` is assigned to a formal parameter in the subroutine, and the subroutine is executed. The subroutine has no power to change the value stored in the variable, `obj`. It only has a copy of that value. However, that value is a reference to an object. Since the subroutine has a reference to the object, it can change the data stored in the object. After the subroutine ends, `obj` still points to the same object, but the data stored **in the object** might have changed. Suppose `x` is a variable of type `int` and `stu` is a variable of type `Student`. Compare:

<pre>void dontChange(int z) { z = 42; }</pre> <p><i>The lines:</i></p> <pre>x = 17; dontChange(x); System.out.println(x);</pre>	<pre>void change(Student s) { s.name = "Fred"; }</pre> <p><i>The lines:</i></p> <pre>stu.name = "Jane"; change(stu); System.out.println(stu.name);</pre>
---	--

output the value 17.

The value of `x` is not changed by the subroutine, which is equivalent to

```
z = x;
z = 42;
```

output the value "Fred".

The value of `stu` is not changed, but `stu.name` is. This is equivalent to

```
s = stu;
s.name = "Fred";
```

5.1.3 Getters and Setters

When writing new classes, it's a good idea to pay attention to the issue of access control. Recall that making a member of a class **public** makes it accessible from anywhere, including from other classes. On the other hand, a **private** member can only be used in the class where it is defined.

In the opinion of many programmers, almost all member variables should be declared **private**. This gives you complete control over what can be done with the variable. Even if the variable itself is private, you can allow other classes to find out what its value is by providing a **public** *accessor method* that returns the value of the variable. For example, if your class contains a **private** member variable, `title`, of type *String*, you can provide a method

```
public String getTitle() {
    return title;
}
```

that returns the value of `title`. By convention, the name of an accessor method for a variable is obtained by capitalizing the name of variable and adding “get” in front of the name. So, for the variable `title`, we get an accessor method named “get” + “Title”, or `getTitle()`. Because of this naming convention, accessor methods are more often referred to as *getter methods*. A getter method provides “read access” to a variable.

You might also want to allow “write access” to a **private** variable. That is, you might want to make it possible for other classes to specify a new value for the variable. This is done with a *setter method*. (If you don't like simple, Anglo-Saxon words, you can use the fancier term *mutator method*.) The name of a setter method should consist of “set” followed by a capitalized copy of the variable's name, and it should have a parameter with the same type as the variable. A setter method for the variable `title` could be written

```
public void setTitle( String newTitle ) {
    title = newTitle;
}
```

It is actually very common to provide both a getter and a setter method for a private member variable. Since this allows other classes both to see and to change the value of the variable, you might wonder why not just make the variable **public**? The reason is that getters and setters are not restricted to simply reading and writing the variable's value. In fact, they can take any action at all. For example, a getter method might keep track of the number of times that the variable has been accessed:

```
public String getTitle() {
    titleAccessCount++; // Increment member variable titleAccessCount.
    return title;
}
```

and a setter method might check that the value that is being assigned to the variable is legal:


```

public void setTitle( String newTitle ) {
    if ( newTitle == null )    // Don't allow null strings as titles!
        title = "(Untitled)"; // Use an appropriate default value instead.
    else
        title = newTitle;
}

```

Even if you can't think of any extra chores to do in a getter or setter method, you might change your mind in the future when you redesign and improve your class. If you've used a getter and setter from the beginning, you can make the modification to your class without affecting any of the classes that use your class. The **private** member variable is not part of the public interface of your class; only the **public** getter and setter methods are. If you **haven't** used `get` and `set` from the beginning, you'll have to contact everyone who uses your class and tell them, "Sorry guys, you'll have to track down every use that you've made of this variable and change your code to use my new `get` and `set` methods instead."

A couple of final notes: Some advanced aspects of Java rely on the naming convention for getter and setter methods, so it's a good idea to follow the convention rigorously. And though I've been talking about using getter and setter methods for a variable, you can define `get` and `set` methods even if there is no variable. A getter and/or setter method defines a *property* of the class, that might or might not correspond to a variable. For example, if a class includes a **public void** instance method with signature `setValue(double)`, then the class has a "property" named `value` of type **double**, and it has this property whether or not the class has a member variable named `value`.

5.2 Constructors and Object Initialization

OBJECT TYPES IN JAVA are very different from the primitive types. Simply declaring a variable whose type is given as a class does not automatically create an object of that class. Objects must be explicitly *constructed*. For the computer, the process of constructing an object means, first, finding some unused memory in the heap that can be used to hold the object and, second, filling in the object's instance variables. As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise some control over what initial values are stored in a new object's instance variables. In many cases, you will also want to do more complicated initialization or bookkeeping every time an object is created.

5.2.1 Initializing Instance Variables

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named `PairOfDice`. An object of this class will represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```

public class PairOfDice {

    public int die1 = 3;    // Number showing on the first die.
    public int die2 = 4;    // Number showing on the second die.

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
    }
}

```

```

        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice

```

The instance variables `die1` and `die2` are initialized to the values 3 and 4 respectively. These initializations are executed whenever a `PairOfDice` object is constructed. It's important to understand when and how this happens. There can be many `PairOfDice` objects. Each time one is created, it gets its own instance variables, and the assignments “`die1 = 3`” and “`die2 = 4`” are executed to fill in the values of those variables. To make this clearer, consider a variation of the `PairOfDice` class:

```

public class PairOfDice {

    public int die1 = (int)(Math.random()*6) + 1;
    public int die2 = (int)(Math.random()*6) + 1;

    public void roll() {
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice

```

Here, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of **static** member variables, of course, the situation is quite different. There is only one copy of a **static** variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (**int**, **double**, etc.) are automatically initialized to zero if you provide no other values; **boolean** variables are initialized to **false**; and **char** variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is **null**. (In particular, since **Strings** are objects, the default initial value for *String* variables is **null**.)

5.2.2 Constructors

Objects are created with the operator, **new**. For example, a program that wants to use a `PairOfDice` object could say:

```

PairOfDice dice;    // Declare a variable of type PairOfDice.

dice = new PairOfDice(); // Construct a new object and store a
                        // reference to it in the variable.

```

In this example, “`new PairOfDice()`” is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, `dice`, so that after the assignment statement is executed, `dice` refers to the newly created object. Part of this expression, “`PairOfDice()`”, looks like a subroutine call, and that is no accident. It is, in fact, a call to a special type of subroutine called a **constructor**. This might puzzle you, since there is no such subroutine in the class definition. However, every class has at least one constructor. If the programmer doesn't write a constructor definition in a class,

then the system will provide a **default constructor** for that class. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other subroutine, with three exceptions. A constructor does not have any return type (not even `void`). The name of the constructor must be the same as the name of the class in which it is defined. The only modifiers that can be used on a constructor definition are the access modifiers `public`, `private`, and `protected`. (In particular, a constructor can't be declared `static`.)

However, a constructor does have a subroutine body of the usual form, a block of statements. There are no restrictions on what statements can be used. And it can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the `PairOfDice` class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

```
public class PairOfDice {

    public int die1;    // Number showing on the first die.
    public int die2;    // Number showing on the second die.

    public PairOfDice(int val1, int val2) {
        // Constructor.  Creates a pair of dice that
        // are initially showing the values val1 and val2.
        die1 = val1;    // Assign specified values
        die2 = val2;    //           to the instance variables.
    }

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The constructor is declared as “`public PairOfDice(int val1, int val2) ...`”, with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression “`new PairOfDice(3,4)`” would create a `PairOfDice` object in which the values of the instance variables `die1` and `die2` are initially 3 and 4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;           // Declare a variable of type PairOfDice.

dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                           //   object that initially shows 1, 1.
```

Now that we've added a constructor to the `PairOfDice` class, we can no longer create an object by saying “`new PairOfDice()`”! The system provides a default constructor for a class **only** if the class definition does not already include a constructor, so there is only one constructor in the class, and it requires two actual parameters. However, this is not a big

problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the `PairOfDice` class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {

    public int die1;    // Number showing on the first die.
    public int die2;    // Number showing on the second die.

    public PairOfDice() {
        // Constructor. Rolls the dice, so that they initially
        // show some random values.
        roll(); // Call the roll() method to roll the dice.
    }

    public PairOfDice(int val1, int val2) {
        // Constructor. Creates a pair of dice that
        // are initially showing the values val1 and val2.
        die1 = val1; // Assign specified values
        die2 = val2; //           to the instance variables.
    }

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Now we have the option of constructing a `PairOfDice` object either with “`new PairOfDice()`” or with “`new PairOfDice(x,y)`”, where `x` and `y` are **int**-valued expressions.

This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation “`(int)(Math.random()*6)+1`”, because it’s done inside the `PairOfDice` class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the `PairOfDice` class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value. This illustrates once again that you can create several instances of the same class:

```
public class RollTwoPairs {

    public static void main(String[] args) {

        PairOfDice firstDice; // Refers to the first pair of dice.
        firstDice = new PairOfDice();

        PairOfDice secondDice; // Refers to the second pair of dice.
        secondDice = new PairOfDice();

        int countRolls; // Counts how many times the two pairs of
                        //      dice have been rolled.

        int total1;      // Total showing on first pair of dice.
        int total2;      // Total showing on second pair of dice.
```

```

        countRolls = 0;

        do { // Roll the two pairs of dice until totals are the same.

            firstDice.roll(); // Roll the first pair of dice.
            total1 = firstDice.die1 + firstDice.die2; // Get total.
            System.out.println("First pair comes up " + total1);

            secondDice.roll(); // Roll the second pair of dice.
            total2 = secondDice.die1 + secondDice.die2; // Get total.
            System.out.println("Second pair comes up " + total2);

            countRolls++; // Count this roll.

            System.out.println(); // Blank line.

        } while (total1 != total2);

        System.out.println("It took " + countRolls
                           + " rolls until the totals were the same.");

    } // end main()

} // end class RollTwoPairs

```

* * *

Constructors are subroutines, but they are subroutines of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any objects have been created. They are more like **static** member subroutines, but they are not and cannot be declared to be **static**. In fact, according to the Java language specification, they are technically not members of the class at all! In particular, constructors are **not** referred to as "methods."

Unlike other subroutines, a constructor can only be called using the **new** operator, in an expression that has the form

```
new <class-name> ( <parameter-list> )
```

where the *<parameter-list>* is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a subroutine call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.
3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.

4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object. You can use this reference to get at the instance variables in that object or to call its instance methods.

* * *

For another example, let's rewrite the `Student` class that was used in Section 1. I'll add a constructor, and I'll also take the opportunity to make the instance variable, `name`, private.

```
public class Student {

    private String name;           // Student's name.
    public double test1, test2, test3; // Grades on three tests.

    Student(String theName) {
        // Constructor for Student objects;
        // provides a name for the Student.
        name = theName;
    }

    public String getName() {
        // Getter method for reading the value of the private
        // instance variable, name.
        return name;
    }

    public double getAverage() {
        // Compute average test grade.
        return (test1 + test2 + test3) / 3;
    }

} // end of class Student
```

An object of type `Student` contains information about some particular student. The constructor in this class has a parameter of type *String*, which specifies the name of that student. Objects of type `Student` can be created with statements such as:

```
std = new Student("John Smith");
std1 = new Student("Mary Jones");
```

In the original version of this class, the value of `name` had to be assigned by a program after it created the object of type `Student`. There was no guarantee that the programmer would always remember to set the `name` properly. In the new version of the class, there is no way to create a `Student` object except by calling the constructor, and that constructor automatically sets the `name`. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.

Another type of guarantee is provided by the `private` modifier. Since the instance variable, `name`, is `private`, there is no way for any part of the program outside the `Student` class to get at the `name` directly. The program sets the value of `name`, indirectly, when it calls the constructor. I've provided a function, `getName()`, that can be used from outside the class to find out the `name` of the student. But I haven't provided any setter method or other way to change the name. Once a student object is created, it keeps the same name as long as it exists.

5.2.3 Garbage Collection

So far, this section has been about creating objects. What about destroying them? In Java, the destruction of objects takes place automatically.

An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this):

```
Student std = new Student("John Smith");  
std = null;
```

In the first line, a reference to a newly created **Student** object is stored in the variable **std**. But in the next line, the value of **std** is changed, and the reference to the **Student** object is gone. In fact, there are now no references whatsoever to that object stored in any variable. So there is no way for the program ever to use the object again. It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called *garbage collection* to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are “garbage.” In the above example, it was very easy to see that the **Student** object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidentally delete an object even though there are still references to that object. This is called a *dangling pointer error*, and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a *memory leak*, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

Because Java uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice.

5.3 Programming with Objects

THERE ARE SEVERAL WAYS in which object-oriented concepts can be applied to the process of designing and writing programs. The broadest of these is *object-oriented analysis and design* which applies an object-oriented methodology to the earliest stages of program development, during which the overall design of a program is created. Here, the idea is to identify things in the problem domain that can be modeled as objects. On another level, object-oriented programming encourages programmers to produce *generalized software components* that can be used in a wide variety of programming projects.

Of course, for the most part, you will experience “generalized software components” by using the standard classes that come along with Java. We begin this section by looking at some built-in classes that are used for creating objects. At the end of the section, we will get back to generalities.

5.3.1 Some Built-in Classes

Although the focus of object-oriented programming is generally on the design and implementation of new classes, it’s important not to forget that the designers of Java have already provided a large number of reusable classes. Some of these classes are meant to be extended to produce new classes, while others can be used directly to create useful objects. A true mastery of Java requires familiarity with a large number of built-in classes—something that takes a lot of time and experience to develop. In the next chapter, we will begin the study of Java’s GUI classes, and you will encounter other built-in classes throughout the remainder of this book. But let’s take a moment to look at a few built-in classes that you might find useful.

A string can be built up from smaller pieces using the `+` operator, but this is not very efficient. If `str` is a *String* and `ch` is a character, then executing the command “`str = str + ch;`” involves creating a whole new string that is a copy of `str`, with the value of `ch` appended onto the end. Copying the string takes some time. Building up a long string letter by letter would require a surprising amount of processing. The class *StringBuffer* makes it possible to be efficient about building up a long string from a number of smaller pieces. To do this, you must make an object belonging to the *StringBuffer* class. For example:

```
StringBuffer buffer = new StringBuffer();
```

(This statement both declares the variable `buffer` and initializes it to refer to a newly created *StringBuffer* object. Combining declaration with initialization was covered in Subsection 4.7.1 and works for objects just as it does for primitive types.)

Like a *String*, a *StringBuffer* contains a sequence of characters. However, it is possible to add new characters onto the end of a *StringBuffer* without making a copy of the data that it already contains. If `x` is a value of any type and `buffer` is the variable defined above, then the command `buffer.append(x)` will add `x`, converted into a string representation, onto the end of the data that was already in the buffer. This command actually modifies the buffer, rather than making a copy, and that can be done efficiently. A long string can be built up in a *StringBuffer* using a sequence of `append()` commands. When the string is complete, the function `buffer.toString()` will return a copy of the string in the buffer as an ordinary value of type *String*. The *StringBuffer* class is in the standard package `java.lang`, so you can use its simple name without importing it.

A number of useful classes are collected in the package `java.util`. For example, this package contains classes for working with collections of objects. We will study these collection classes in Chapter 10. Another class in this package, `java.util.Date`, is used to represent times. When a *Date* object is constructed without parameters, the result represents the current date and time, so an easy way to display this information is:

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

Of course, to use the *Date* class in this way, you must make it available by importing it with one of the statements “`import java.util.Date;`” or “`import java.util.*;`” at the beginning of your program. (See Subsection 4.5.3 for a discussion of packages and `import`.)

I will also mention the class `java.util.Random`. An object belonging to this class is a *source* of random numbers (or, more precisely pseudorandom numbers). The standard function

`Math.random()` uses one of these objects behind the scenes to generate its random numbers. An object of type `Random` can generate random integers, as well as random real numbers. If `randGen` is created with the command:

```
Random randGen = new Random();
```

and if `N` is a positive integer, then `randGen.nextInt(N)` generates a random integer in the range from 0 to `N-1`. For example, this makes it a little easier to roll a pair of dice. Instead of saying “`die1 = (int)(6*Math.random()+1);`”, one can say “`die1 = randGen.nextInt(6)+1;`”. (Since you also have to import the class `java.util.Random` and create the `Random` object, you might not agree that it is actually easier.) An object of type `Random` can also be used to generate so-called Gaussian distributed random real numbers.

The main point here, again, is that many problems have already been solved, and the solutions are available in Java’s standard classes. If you are faced with a task that looks like it should be fairly common, it might be worth looking through a Java reference to see whether someone has already written a class that you can use.

5.3.2 Wrapper Classes and Autoboxing

We have already encountered the classes *Double* and *Integer* in Subsection 2.5.7. These classes contain the `static` methods `Double.parseDouble` and `Integer.parseInt` that are used to convert strings to numerical values. We have also encountered the *Character* class in some examples, and static methods such as `Character.isLetter`, which can be used to test whether a given value of type `char` is a letter. There is a similar class for each of the other primitive types, *Long*, *Short*, *Byte*, *Float*, and *Boolean*. These classes are called *wrapper classes*. Although they contain useful `static` members, they have another use as well: They are used for creating objects that represent primitive type values.

Remember that the primitive types are not classes, and values of primitive type are not objects. However, sometimes it’s useful to treat a primitive value as if it were an object. You can’t do that literally, but you can “wrap” the primitive type value in an object belonging to one of the wrapper classes.

For example, an object of type *Double* contains a single instance variable, of type **double**. The object is a wrapper for the **double** value. For example, you can create an object that wraps the **double** value `6.0221415e23` with

```
Double d = new Double(6.0221415e23);
```

The value of `d` contains the same information as the value of type **double**, but it is an object. If you want to retrieve the **double** value that is wrapped in the object, you can call the function `d.doubleValue()`. Similarly, you can wrap an **int** in an object of type *Integer*, a **boolean** value in an object of type *Boolean*, and so on. (As an example of where this would be useful, the collection classes that will be studied in Chapter 10 can only hold objects. If you want to add a primitive type value to a collection, it has to be put into a wrapper object first.)

In Java 5.0, wrapper classes have become easier to use. Java 5.0 introduced automatic conversion between a primitive type and the corresponding wrapper class. For example, if you use a value of type **int** in a context that requires an object of type *Integer*, the **int** will automatically be wrapped in an *Integer* object. For example, you can say

```
Integer answer = 42;
```

and the computer will silently read this as if it were

```
Integer answer = new Integer(42);
```

This is called *autoboxing*. It works in the other direction, too. For example, if `d` refers to an object of type `Double`, you can use `d` in a numerical expression such as `2*d`. The **double** value inside `d` is automatically *unboxed* and multiplied by 2. Autoboxing and unboxing also apply to subroutine calls. For example, you can pass an actual parameter of type **int** to a subroutine that has a formal parameter of type *Integer*. In fact, autoboxing and unboxing make it possible in many circumstances to ignore the difference between primitive types and objects.

* * *

The wrapper classes contain a few other things that deserve to be mentioned. *Integer*, for example, contains constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, which are equal to the largest and smallest possible values of type **int**, that is, to -2147483648 and 2147483647 respectively. It's certainly easier to remember the names than the numerical values. There are similar named constants in *Long*, *Short*, and *Byte*. *Double* and *Float* also have constants named `MIN_VALUE` and `MAX_VALUE`. `MAX_VALUE` still gives the largest number that can be represented in the given type, but `MIN_VALUE` represents the smallest possible **positive** value. For type **double**, `Double.MIN_VALUE` is 4.9 times 10^{-324} . Since **double** values have only a finite accuracy, they can't get arbitrarily close to zero. This is the closest they can get without actually being equal to zero.

The class *Double* deserves special mention, since **doubles** are so much more complicated than integers. The encoding of real numbers into values of type **double** has room for a few special values that are not real numbers at all in the mathematical sense. These values are given by named constants in class *Double*: `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, and `Double.NaN`. The infinite values can occur as the values of certain mathematical expressions. For example, dividing a positive number by zero will give the result `Double.POSITIVE_INFINITY`. (It's even more complicated than this, actually, because the **double** type includes a value called “negative zero”, written `-0.0`. Dividing a positive number by negative zero gives `Double.NEGATIVE_INFINITY`.) You also get `Double.POSITIVE_INFINITY` whenever the mathematical value of an expression is greater than `Double.MAX_VALUE`. For example, `1e200*1e200` is considered to be infinite. The value `Double.NaN` is even more interesting. “NaN” stands for *Not a Number*, and it represents an undefined value such as the square root of a negative number or the result of dividing zero by zero. Because of the existence of `Double.NaN`, no mathematical operation on real numbers will ever throw an exception; it simply gives `Double.NaN` as the result.

You can test whether a value, `x`, of type **double** is infinite or undefined by calling the boolean-valued static functions `Double.isInfinite(x)` and `Double.isNaN(x)`. (It's especially important to use `Double.isNaN()` to test for undefined values, because `Double.NaN` has really weird behavior when used with relational operators such as `==`. In fact, the values of `x == Double.NaN` and `x != Double.NaN` are always **both false**—no matter what the value of `x` is—so you can't use these expressions to test whether `x` is `Double.NaN`.)

5.3.3 The class “Object”

We have already seen that one of the major features of object-oriented programming is the ability to create subclasses of a class. The subclass inherits all the properties or behaviors of the class, but can modify and add to what it inherits. In Section 5.5, you'll learn how to create subclasses. What you don't know yet is that **every** class in Java (with just one exception) is a subclass of some other class. If you create a class and don't explicitly make it a subclass of

some other class, then it automatically becomes a subclass of the special class named *Object*. (*Object* is the one class that is not a subclass of any other class.)

Class *Object* defines several instance methods that are inherited by every other class. These methods can be used with any object whatsoever. I will mention just one of them here. You will encounter more of them later in the book.

The instance method `toString()` in class *Object* returns a value of type *String* that is supposed to be a string representation of the object. You've already used this method implicitly, any time you've printed out an object or concatenated an object onto a string. When you use an object in a context that requires a string, the object is automatically converted to type *String* by calling its `toString()` method.

The version of `toString` that is defined in *Object* just returns the name of the class that the object belongs to, concatenated with a code number called the *hash code* of the object; this is not very useful. When you create a class, you can write a new `toString()` method for it, which will replace the inherited version. For example, we might add the following method to any of the *PairOfDice* classes from the previous section:

```
public String toString() {
    // Return a String representation of a pair of dice, where die1
    // and die2 are instance variables containing the numbers that are
    // showing on the two dice.
    if (die1 == die2)
        return "double " + die1;
    else
        return die1 + " and " + die2;
}
```

If `dice` refers to a *PairOfDice* object, then `dice.toString()` will return strings such as “3 and 6”, “5 and 1”, and “double 2”, depending on the numbers showing on the dice. This method would be used automatically to convert `dice` to type *String* in a statement such as

```
System.out.println( "The dice came up " + dice );
```

so this statement might output, “The dice came up 5 and 1” or “The dice came up double 2”. You'll see another example of a `toString()` method in the next section.

5.3.4 Object-oriented Analysis and Design

Every programmer builds up a stock of techniques and expertise expressed as snippets of code that can be reused in new programs using the tried-and-true method of cut-and-paste: The old code is physically copied into the new program and then edited to customize it as necessary. The problem is that the editing is error-prone and time-consuming, and the whole enterprise is dependent on the programmer's ability to pull out that particular piece of code from last year's project that looks like it might be made to fit. (On the level of a corporation that wants to save money by not reinventing the wheel for each new project, just keeping track of all the old wheels becomes a major task.)

Well-designed classes are software components that can be reused without editing. A well-designed class is not carefully crafted to do a particular job in a particular program. Instead, it is crafted to model some particular type of object or a single coherent concept. Since objects and concepts can recur in many problems, a well-designed class is likely to be reusable without modification in a variety of projects.

Furthermore, in an object-oriented programming language, it is possible to make *subclasses* of an existing class. This makes classes even more reusable. If a class needs to be customized, a subclass can be created, and additions or modifications can be made in the subclass without making any changes to the original class. This can be done even if the programmer doesn't have access to the source code of the class and doesn't know any details of its internal, hidden implementation.

* * *

The *PairOfDice* class in the previous section is already an example of a generalized software component, although one that could certainly be improved. The class represents a single, coherent concept, "a pair of dice." The instance variables hold the data relevant to the state of the dice, that is, the number showing on each of the dice. The instance method represents the behavior of a pair of dice, that is, the ability to be rolled. This class would be reusable in many different programming projects.

On the other hand, the *Student* class from the previous section is not very reusable. It seems to be crafted to represent students in a particular course where the grade will be based on three tests. If there are more tests or quizzes or papers, it's useless. If there are two people in the class who have the same name, we are in trouble (one reason why numerical student ID's are often used). Admittedly, it's much more difficult to develop a general-purpose student class than a general-purpose pair-of-dice class. But this particular *Student* class is good mostly as an example in a programming textbook.

* * *

A large programming project goes through a number of stages, starting with *specification* of the problem to be solved, followed by *analysis* of the problem and *design* of a program to solve it. Then comes *coding*, in which the program's design is expressed in some actual programming language. This is followed by *testing* and *debugging* of the program. After that comes a long period of *maintenance*, which means fixing any new problems that are found in the program and modifying it to adapt it to changing requirements. Together, these stages form what is called the *software life cycle*. (In the real world, the ideal of consecutive stages is seldom if ever achieved. During the analysis stage, it might turn out that the specifications are incomplete or inconsistent. A problem found during testing requires at least a brief return to the coding stage. If the problem is serious enough, it might even require a new design. Maintenance usually involves redoing some of the work from previous stages. . . .)

Large, complex programming projects are only likely to succeed if a careful, systematic approach is adopted during all stages of the software life cycle. The systematic approach to programming, using accepted principles of good design, is called *software engineering*. The software engineer tries to efficiently construct programs that verifiably meet their specifications and that are easy to modify if necessary. There is a wide range of "methodologies" that can be applied to help in the systematic design of programs. (Most of these methodologies seem to involve drawing little boxes to represent program components, with labeled arrows to represent relationships among the boxes.)

We have been discussing object orientation in programming languages, which is relevant to the coding stage of program development. But there are also object-oriented methodologies for analysis and design. The question in this stage of the software life cycle is, How can one discover or invent the overall structure of a program? As an example of a rather simple object-oriented approach to analysis and design, consider this advice: Write down a description of the problem. Underline all the nouns in that description. The nouns should be considered as candidates for becoming classes or objects in the program design. Similarly, underline all the verbs. These

are candidates for methods. This is your starting point. Further analysis might uncover the need for more classes and methods, and it might reveal that subclassing can be used to take advantage of similarities among classes.

This is perhaps a bit simple-minded, but the idea is clear and the general approach can be effective: Analyze the problem to discover the concepts that are involved, and create classes to represent those concepts. The design should arise from the problem itself, and you should end up with a program whose structure reflects the structure of the problem in a natural way.

5.4 Programming Example: Card, Hand, Deck

In this section, we look at some specific examples of object-oriented design in a domain that is simple enough that we have a chance of coming up with something reasonably reusable. Consider card games that are played with a standard deck of playing cards (a so-called “poker” deck, since it is used in the game of poker).

5.4.1 Designing the classes

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players’ hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, . . . , king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives. If we look for nouns in this description, there are several candidates for objects: game, player, hand, card, deck, value, and suit. Of these, the value and the suit of a card are simple values, and they will just be represented as instance variables in a *Card* object. In a complete program, the other five nouns might be represented by classes. But let’s work on the ones that are most obviously reusable: card, hand, and deck.

If we look for verbs in the description of a card game, we see that we can shuffle a deck and deal a card from a deck. This gives us two candidates for instance methods in a *Deck* class: `shuffle()` and `dealCard()`. Cards can be added to and removed from hands. This gives two candidates for instance methods in a *Hand* class: `addCard()` and `removeCard()`. Cards are relatively passive things, but we need to be able to determine their suits and values. We will discover more instance methods as we go along.

First, we’ll design the deck class in detail. When a deck of cards is first created, it contains 52 cards in some standard order. The *Deck* class will need a constructor to create a new deck. The constructor needs no parameters because any new deck is the same as any other. There will be an instance method called `shuffle()` that will rearrange the 52 cards into a random order. The `dealCard()` instance method will get the next card from the deck. This will be a function with a return type of *Card*, since the caller needs to know what card is being dealt. It has no parameters—when you deal the next card from the deck, you don’t provide any information to the deck; you just get the next card, whatever it is. What will happen if there are no more cards in the deck when its `dealCard()` method is called? It should probably be considered an error to try to deal a card from an empty deck, so the deck can throw an exception in that case. But this raises another question: How will the rest of the program know whether the deck is empty? Of course, the program could keep track of how many cards it has used. But the deck itself should know how many cards it has left, so the program should just be able to ask the deck object. We can make this possible by specifying another instance method, `cardsLeft()`, that returns the number of cards remaining in the deck. This leads to a full specification of all

the subroutines in the *Deck* class:

Constructor and instance methods in class *Deck*:

```
public Deck()
    // Constructor. Create an unshuffled deck of cards.

public void shuffle()
    // Put all the used cards back into the deck,
    // and shuffle it into a random order.

public int cardsLeft()
    // As cards are dealt from the deck, the number of
    // cards left decreases. This function returns the
    // number of cards that are still left in the deck.

public Card dealCard()
    // Deals one card from the deck and returns it.
    // Throws an exception if no more cards are left.
```

This is everything you need to know in order to use the *Deck* class. Of course, it doesn't tell us how to write the class. This has been an exercise in design, not in programming. In fact, writing the class involves a programming technique, arrays, which will not be covered until Chapter 7. Nevertheless, you can look at the source code, *Deck.java*, if you want. Even though you won't understand the implementation, the Javadoc comments give you all the information that you need to understand the interface. With this information, you can use the class in your programs without understanding the implementation.

We can do a similar analysis for the *Hand* class. When a hand object is first created, it has no cards in it. An `addCard()` instance method will add a card to the hand. This method needs a parameter of type *Card* to specify which card is being added. For the `removeCard()` method, a parameter is needed to specify which card to remove. But should we specify the card itself ("Remove the ace of spades"), or should we specify the card by its position in the hand ("Remove the third card in the hand")? Actually, we don't have to decide, since we can allow for both options. We'll have two `removeCard()` instance methods, one with a parameter of type *Card* specifying the card to be removed and one with a parameter of type `int` specifying the position of the card in the hand. (Remember that you can have two methods in a class with the same name, provided they have different types of parameters.) Since a hand can contain a variable number of cards, it's convenient to be able to ask a hand object how many cards it contains. So, we need an instance method `getCardCount()` that returns the number of cards in the hand. When I play cards, I like to arrange the cards in my hand so that cards of the same value are next to each other. Since this is a generally useful thing to be able to do, we can provide instance methods for sorting the cards in the hand. Here is a full specification for a reusable *Hand* class:

Constructor and instance methods in class *Hand*:

```
public Hand() {
    // Create a Hand object that is initially empty.

public void clear() {
    // Discard all cards from the hand, making the hand empty.

public void addCard(Card c) {
    // Add the card c to the hand. c should be non-null.
    // If c is null, a NullPointerException is thrown.
```

```

public void removeCard(Card c) {
    // If the specified card is in the hand, it is removed.

public void removeCard(int position) {
    // Remove the card in the specified position from the
    // hand. Cards are numbered counting from zero. If
    // the specified position does not exist, then an
    // exception is thrown.

public int getCardCount() {
    // Return the number of cards in the hand.

public Card getCard(int position) {
    // Get the card from the hand in given position, where
    // positions are numbered starting from 0. If the
    // specified position is not the position number of
    // a card in the hand, an exception is thrown.

public void sortBySuit() {
    // Sorts the cards in the hand so that cards of the same
    // suit are grouped together, and within a suit the cards
    // are sorted by value. Note that aces are considered
    // to have the lowest value, 1.

public void sortByValue() {
    // Sorts the cards in the hand so that cards are sorted into
    // order of increasing value. Cards with the same value
    // are sorted by suit. Note that aces are considered
    // to have the lowest value.

```

Again, you don't yet know enough to implement this class. But given the source code, *Hand.java*, you can use the class in your own programming projects.

5.4.2 The Card Class

We **have** covered enough material to write a *Card* class. The class will have a constructor that specifies the value and suit of the card that is being created. There are four suits, which can be represented by the integers 0, 1, 2, and 3. It would be tough to remember which number represents which suit, so I've defined named constants in the *Card* class to represent the four possibilities. For example, *Card.SPADES* is a constant that represents the suit, spades. (These constants are declared to be **public final static ints**. It might be better to use an enumerated type, but for now we will stick to integer-valued constants. I'll return to the question of using enumerated types in this example at the end of the chapter.) The possible values of a card are the numbers 1, 2, ..., 13, with 1 standing for an ace, 11 for a jack, 12 for a queen, and 13 for a king. Again, I've defined some named constants to represent the values of aces and face cards. (When you read the *Card* class, you'll see that I've also added support for Jokers.)

A *Card* object can be constructed knowing the value and the suit of the card. For example, we can call the constructor with statements such as:

```

card1 = new Card( Card.ACE, Card.SPADES ); // Construct ace of spades.
card2 = new Card( 10, Card.DIAMONDS );    // Construct 10 of diamonds.
card3 = new Card( v, s ); // This is OK, as long as v and s
                        // are integer expressions.

```

A *Card* object needs instance variables to represent its value and suit. I've made these `private` so that they cannot be changed from outside the class, and I've provided getter methods `getSuit()` and `getValue()` so that it will be possible to discover the suit and value from outside the class. The instance variables are initialized in the constructor, and are never changed after that. In fact, I've declared the instance variables `suit` and `value` to be `final`, since they are never changed after they are initialized. (An instance variable can be declared `final` provided it is either given an initial value in its declaration or is initialized in every constructor in the class.)

Finally, I've added a few convenience methods to the class to make it easier to print out cards in a human-readable form. For example, I want to be able to print out the suit of a card as the word "Diamonds", rather than as the meaningless code number 2, which is used in the class to represent diamonds. Since this is something that I'll probably have to do in many programs, it makes sense to include support for it in the class. So, I've provided instance methods `getSuitAsString()` and `getValueAsString()` to return string representations of the suit and value of a card. Finally, I've defined the instance method `toString()` to return a string with both the value and suit, such as "Queen of Hearts". Recall that this method will be used whenever a *Card* needs to be converted into a *String*, such as when the card is concatenated onto a string with the `+` operator. Thus, the statement

```
System.out.println( "Your card is the " + card );
```

is equivalent to

```
System.out.println( "Your card is the " + card.toString() );
```

If the card is the queen of hearts, either of these will print out "Your card is the Queen of Hearts".

Here is the complete *Card* class. It is general enough to be highly reusable, so the work that went into designing, writing, and testing it pays off handsomely in the long run.

```
/**
 * An object of type Card represents a playing card from a
 * standard Poker deck, including Jokers. The card has a suit, which
 * can be spades, hearts, diamonds, clubs, or joker. A spade, heart,
 * diamond, or club has one of the 13 values: ace, 2, 3, 4, 5, 6, 7,
 * 8, 9, 10, jack, queen, or king. Note that "ace" is considered to be
 * the smallest value. A joker can also have an associated value;
 * this value can be anything and can be used to keep track of several
 * different jokers.
 */

public class Card {

    public final static int SPADES = 0;    // Codes for the 4 suits, plus Joker.
    public final static int HEARTS = 1;
    public final static int DIAMONDS = 2;
    public final static int CLUBS = 3;
    public final static int JOKER = 4;

    public final static int ACE = 1;        // Codes for the non-numeric cards.
    public final static int JACK = 11;      // Cards 2 through 10 have their
    public final static int QUEEN = 12;     // numerical values for their codes.
    public final static int KING = 13;

    /**
```



```

    * This card's suit, one of the constants SPADES, HEARTS, DIAMONDS,
    * CLUBS, or JOKER. The suit cannot be changed after the card is
    * constructed.
    */
private final int suit;

/**
 * The card's value. For a normal cards, this is one of the values
 * 1 through 13, with 1 representing ACE. For a JOKER, the value
 * can be anything. The value cannot be changed after the card
 * is constructed.
 */
private final int value;

/**
 * Creates a Joker, with 1 as the associated value. (Note that
 * "new Card()" is equivalent to "new Card(1,Card.JOKER)".)
 */
public Card() {
    suit = JOKER;
    value = 1;
}

/**
 * Creates a card with a specified suit and value.
 * @param theValue the value of the new card. For a regular card (non-joker),
 * the value must be in the range 1 through 13, with 1 representing an Ace.
 * You can use the constants Card.ACE, Card.JACK, Card.QUEEN, and Card.KING.
 * For a Joker, the value can be anything.
 * @param theSuit the suit of the new card. This must be one of the values
 * Card.SPADES, Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or Card.JOKER.
 * @throws IllegalArgumentException if the parameter values are not in the
 * permissible ranges
 */
public Card(int theValue, int theSuit) {
    if (theSuit != SPADES && theSuit != HEARTS && theSuit != DIAMONDS &&
        theSuit != CLUBS && theSuit != JOKER)
        throw new IllegalArgumentException("Illegal playing card suit");
    if (theSuit != JOKER && (theValue < 1 || theValue > 13))
        throw new IllegalArgumentException("Illegal playing card value");
    value = theValue;
    suit = theSuit;
}

/**
 * Returns the suit of this card.
 * @return the suit, which is one of the constants Card.SPADES,
 * Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or Card.JOKER
 */
public int getSuit() {
    return suit;
}

/**
 * Returns the value of this card.
 * @return the value, which is one the numbers 1 through 13, inclusive for

```

```

    * a regular card, and which can be any value for a Joker.
    */
    public int getValue() {
        return value;
    }

    /**
     * Returns a String representation of the card's suit.
     * @return one of the strings "Spades", "Hearts", "Diamonds", "Clubs"
     * or "Joker".
     */
    public String getSuitAsString() {
        switch ( suit ) {
            case SPADES:    return "Spades";
            case HEARTS:    return "Hearts";
            case DIAMONDS: return "Diamonds";
            case CLUBS:     return "Clubs";
            default:        return "Joker";
        }
    }

    /**
     * Returns a String representation of the card's value.
     * @return for a regular card, one of the strings "Ace", "2",
     * "3", ..., "10", "Jack", "Queen", or "King". For a Joker, the
     * string is always a numerical.
     */
    public String getValueAsString() {
        if (suit == JOKER)
            return "" + value;
        else {
            switch ( value ) {
                case 1:    return "Ace";
                case 2:    return "2";
                case 3:    return "3";
                case 4:    return "4";
                case 5:    return "5";
                case 6:    return "6";
                case 7:    return "7";
                case 8:    return "8";
                case 9:    return "9";
                case 10:   return "10";
                case 11:   return "Jack";
                case 12:   return "Queen";
                default:   return "King";
            }
        }
    }

    /**
     * Returns a string representation of this card, including both
     * its suit and its value (except that for a Joker with value 1,
     * the return value is just "Joker"). Sample return values
     * are: "Queen of Hearts", "10 of Diamonds", "Ace of Spades",
     * "Joker", "Joker #2"

```

```

        */
        public String toString() {
            if (suit == JOKER) {
                if (value == 1)
                    return "Joker";
                else
                    return "Joker #" + value;
            }
            else
                return getValueAsString() + " of " + getSuitAsString();
        }
    } // end class Card

```

5.4.3 Example: A Simple Card Game

I will finish this section by presenting a complete program that uses the *Card* and *Deck* classes. The program lets the user play a very simple card game called HighLow. A deck of cards is shuffled, and one card is dealt from the deck and shown to the user. The user predicts whether the next card from the deck will be higher or lower than the current card. If the user predicts correctly, then the next card from the deck becomes the current card, and the user makes another prediction. This continues until the user makes an incorrect prediction. The number of correct predictions is the user's score.

My program has a subroutine that plays one game of HighLow. This subroutine has a return value that represents the user's score in the game. The `main()` routine lets the user play several games of HighLow. At the end, it reports the user's average score.

I won't go through the development of the algorithms used in this program, but I encourage you to read it carefully and make sure that you understand how it works. Note in particular that the subroutine that plays one game of HighLow returns the user's score in the game as its return value. This gets the score back to the main program, where it is needed. Here is the program:

```

/**
 * This program lets the user play HighLow, a simple card game
 * that is described in the output statements at the beginning of
 * the main() routine. After the user plays several games,
 * the user's average score is reported.
 */

public class HighLow {

    public static void main(String[] args) {

        System.out.println("This program lets you play the simple card game,");
        System.out.println("HighLow. A card is dealt from a deck of cards.");
        System.out.println("You have to predict whether the next card will be");
        System.out.println("higher or lower. Your score in the game is the");
        System.out.println("number of correct predictions you make before");
        System.out.println("you guess wrong.");
        System.out.println();

        int gamesPlayed = 0;    // Number of games user has played.
        int sumOfScores = 0;    // The sum of all the scores from

```

```

                                //      all the games played.
double averageScore;           // Average score, computed by dividing
                                //      sumOfScores by gamesPlayed.
boolean playAgain;             // Record user's response when user is
                                //      asked whether he wants to play
                                //      another game.

do {
    int scoreThisGame;          // Score for one game.
    scoreThisGame = play();     // Play the game and get the score.
    sumOfScores += scoreThisGame;
    gamesPlayed++;
    TextIO.put("Play again? ");
    playAgain = TextIO.getlnBoolean();
} while (playAgain);

averageScore = ((double)sumOfScores) / gamesPlayed;

System.out.println();
System.out.println("You played " + gamesPlayed + " games.");
System.out.printf("Your average score was %1.3f.\n", averageScore);
} // end main()

/**
 * Lets the user play one game of HighLow, and returns the
 * user's score on that game. The score is the number of
 * correct guesses that the user makes.
 */
private static int play() {
    Deck deck = new Deck(); // Get a new deck of cards, and
                            //      store a reference to it in
                            //      the variable, deck.

    Card currentCard; // The current card, which the user sees.

    Card nextCard; // The next card in the deck. The user tries
                  //      to predict whether this is higher or lower
                  //      than the current card.

    int correctGuesses ; // The number of correct predictions the
                        //      user has made. At the end of the game,
                        //      this will be the user's score.

    char guess; // The user's guess. 'H' if the user predicts that
               //      the next card will be higher, 'L' if the user
               //      predicts that it will be lower.

    deck.shuffle(); // Shuffle the deck into a random order before
                  //      starting the game.

    correctGuesses = 0;
    currentCard = deck.dealCard();
    TextIO.putln("The first card is the " + currentCard);

    while (true) { // Loop ends when user's prediction is wrong.

        /* Get the user's prediction, 'H' or 'L' (or 'h' or 'l'). */

```

```

TextIO.put("Will the next card be higher (H) or lower (L)? ");
do {
    guess = TextIO.getlnChar();
    guess = Character.toUpperCase(guess);
    if (guess != 'H' && guess != 'L')
        TextIO.put("Please respond with H or L: ");
} while (guess != 'H' && guess != 'L');

/* Get the next card and show it to the user. */
nextCard = deck.dealCard();
TextIO.putln("The next card is " + nextCard);

/* Check the user's prediction. */
if (nextCard.getValue() == currentCard.getValue()) {
    TextIO.putln("The value is the same as the previous card.");
    TextIO.putln("You lose on ties. Sorry!");
    break; // End the game.
}
else if (nextCard.getValue() > currentCard.getValue()) {
    if (guess == 'H') {
        TextIO.putln("Your prediction was correct.");
        correctGuesses++;
    }
    else {
        TextIO.putln("Your prediction was incorrect.");
        break; // End the game.
    }
}
else { // nextCard is lower
    if (guess == 'L') {
        TextIO.putln("Your prediction was correct.");
        correctGuesses++;
    }
    else {
        TextIO.putln("Your prediction was incorrect.");
        break; // End the game.
    }
}

/* To set up for the next iteration of the loop, the nextCard
   becomes the currentCard, since the currentCard has to be
   the card that the user sees, and the nextCard will be
   set to the next card in the deck after the user makes
   his prediction. */
currentCard = nextCard;
TextIO.putln();
TextIO.putln("The card is " + currentCard);

} // end of while loop

TextIO.putln();
TextIO.putln("The game is over.");
TextIO.putln("You made " + correctGuesses

```

```

                                + " correct predictions.");
    TextIO.putln();
    return correctGuesses;
} // end play()

} // end class

```

5.5 Inheritance, Polymorphism, and Abstract Classes

A CLASS REPRESENTS A SET OF OBJECTS which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming—the idea that really distinguishes it from traditional programming—is to allow classes to express the similarities among objects that share **some**, but not all, of their structure and behavior. Such similarities can be expressed using *inheritance* and *polymorphism*.

5.5.1 Extending Existing Classes

The topics covered later in this section are relatively advanced aspects of object-oriented programming. Any programmer should know what is meant by subclass, inheritance, and polymorphism. However, it will probably be a while before you actually do anything with inheritance except for extending classes that already exist. In the first part of this section, we look at how that is done.

In day-to-day programming, especially for programmers who are just beginning to work with objects, subclassing is used mainly in one situation: There is an existing class that can be adapted with a few changes or additions. This is much more common than designing groups of classes and subclasses from scratch. The existing class can be *extended* to make a subclass. The syntax for this is

```

public class <subclass-name> extends <existing-class-name> {
    .
    .    // Changes and additions.
    .
}

```

As an example, suppose you want to write a program that plays the card game, Blackjack. You can use the *Card*, *Hand*, and *Deck* classes developed in Section 5.4. However, a hand in the game of Blackjack is a little different from a hand of cards in general, since it must be possible to compute the “value” of a Blackjack hand according to the rules of the game. The rules are as follows: The value of a hand is obtained by adding up the values of the cards in the hand. The value of a numeric card such as a three or a ten is its numerical value. The value of a Jack, Queen, or King is 10. The value of an Ace can be either 1 or 11. An Ace should be counted as 11 unless doing so would put the total value of the hand over 21. Note that this means that the second, third, or fourth Ace in the hand will always be counted as 1.

One way to handle this is to extend the existing *Hand* class by adding a method that computes the Blackjack value of the hand. Here’s the definition of such a class:

```

public class BlackjackHand extends Hand {

    /**
     * Computes and returns the value of this hand in the game
     * of Blackjack.
     */
    public int getBlackjackValue() {

        int val;          // The value computed for the hand.
        boolean ace;      // This will be set to true if the
                        // hand contains an ace.
        int cards;        // Number of cards in the hand.

        val = 0;
        ace = false;
        cards = getCardCount();

        for ( int i = 0; i < cards; i++ ) {
            // Add the value of the i-th card in the hand.
            Card card;    // The i-th card;
            int cardVal;  // The blackjack value of the i-th card.
            card = getCard(i);
            cardVal = card.getValue(); // The normal value, 1 to 13.
            if (cardVal > 10) {
                cardVal = 10; // For a Jack, Queen, or King.
            }
            if (cardVal == 1) {
                ace = true; // There is at least one ace.
            }
            val = val + cardVal;
        }

        // Now, val is the value of the hand, counting any ace as 1.
        // If there is an ace, and if changing its value from 1 to
        // 11 would leave the score less than or equal to 21,
        // then do so by adding the extra 10 points to val.

        if ( ace == true && val + 10 <= 21 )
            val = val + 10;

        return val;
    } // end getBlackjackValue()

} // end class BlackjackHand

```

Since *BlackjackHand* is a subclass of *Hand*, an object of type *BlackjackHand* contains all the instance variables and instance methods defined in *Hand*, plus the new instance method named `getBlackjackValue()`. For example, if `bjh` is a variable of type *BlackjackHand*, then the following are all legal: `bjh.getCardCount()`, `bjh.removeCard(0)`, and `bjh.getBlackjackValue()`. The first two methods are defined in *Hand*, but are inherited by *BlackjackHand*.

Inherited variables and methods from the *Hand* class can also be used in the definition of *BlackjackHand* (except for any that are declared to be `private`, which prevents access even by subclasses). The statement “`cards = getCardCount();`” in the above definition of `getBlackjackValue()` calls the instance method `getCardCount()`, which was defined in *Hand*.

Extending existing classes is an easy way to build on previous work. We'll see that many standard classes have been written specifically to be used as the basis for making subclasses.

* * *

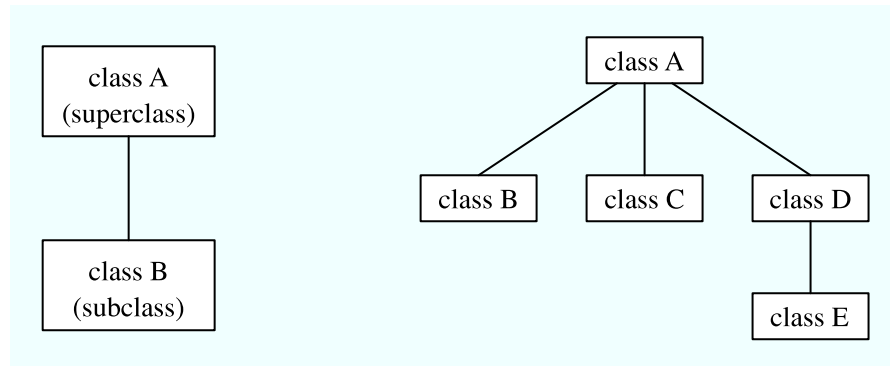
Access modifiers such as **public** and **private** are used to control access to members of a class. There is one more access modifier, **protected**, that comes into the picture when subclasses are taken into consideration. When **protected** is applied as an access modifier to a method or member variable in a class, that member can be used in subclasses—direct or indirect—of the class in which it is defined, but it cannot be used in non-subclasses. (There is one exception: A **protected** member can also be accessed by any class in the same package as the class that contains the protected member. Recall that using no access modifier makes a member accessible to classes in the same package, and nowhere else. Using the **protected** modifier is strictly more liberal than using no modifier at all: It allows access from classes in the same package and from **subclasses** that are not in the same package.)

When you declare a method or member variable to be **protected**, you are saying that it is part of the implementation of the class, rather than part of the public interface of the class. However, you are allowing subclasses to use and modify that part of the implementation.

For example, consider a *PairOfDice* class that has instance variables **die1** and **die2** to represent the numbers appearing on the two dice. We could make those variables **private** to make it impossible to change their values from outside the class, while still allowing read access through getter methods. However, if we think it possible that *PairOfDice* will be used to create subclasses, we might want to make it possible for subclasses to change the numbers on the dice. For example, a *GraphicalDice* subclass that draws the dice might want to change the numbers at other times besides when the dice are rolled. In that case, we could make **die1** and **die2** **protected**, which would allow the subclass to change their values without making them public to the rest of the world. (An even better idea would be to define **protected** setter methods for the variables. A setter method could, for example, ensure that the value that is being assigned to the variable is in the legal range 1 through 6.)

5.5.2 Inheritance and Class Hierarchy

The term *inheritance* refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a **subclass** of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a **superclass** of class B. (Sometimes the terms **derived class** and **base class** are used instead of subclass and superclass; this is the common terminology in C++.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass, as shown here on the left.



In Java, to create a class named “B” as a subclass of a class named “A”, you would write

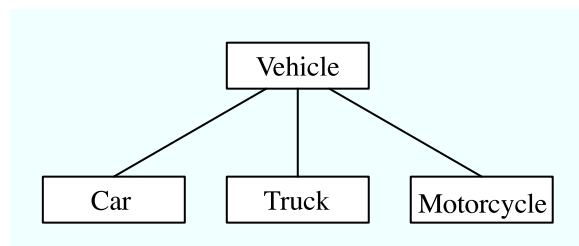
```

class B extends A {
    .
    . // additions to, and modifications of,
    . // stuff inherited from class A
    .
}
  
```

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as “sibling classes,” share some structures and behaviors—namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram shown on the right, above, classes B, C, and D are sibling classes. Inheritance can also extend over several “generations” of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes forms a small *class hierarchy*.

5.5.3 Example: Vehicles

Let’s look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named *Vehicle* to represent all types of vehicles. Since cars, trucks, and motorcycles are types of vehicles, they would be represented by subclasses of the *Vehicle* class, as shown in this class hierarchy diagram:



The *Vehicle* class would include instance variables such as `registrationNumber` and `owner` and instance methods such as `transferOwnership()`. These are variables and methods common to all vehicles. The three subclasses of *Vehicle*—*Car*, *Truck*, and *Motorcycle*—could then be used to hold variables and methods specific to particular types of vehicles. The *Car* class might add an instance variable `numberOfDoors`, the *Truck* class might have `numberOfAxes`,

and the *Motorcycle* class could have a boolean variable `hasSidecar`. (Well, it could in theory at least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in a Java program would look, in outline, like this (although in practice, they would probably be `public` classes, defined in separate files):

```
class Vehicle {
    int registrationNumber;
    Person owner; // (Assuming that a Person class has been defined!)
    void transferOwnership(Person newOwner) {
        . . .
    }
    . . .
}

class Car extends Vehicle {
    int numberOfDoors;
    . . .
}

class Truck extends Vehicle {
    int numberOfAxles;
    . . .
}

class Motorcycle extends Vehicle {
    boolean hasSidecar;
    . . .
}
```

Suppose that `myCar` is a variable of type *Car* that has been declared and initialized with the statement

```
Car myCar = new Car();
```

Given this declaration, a program could refer to `myCar.numberOfDoors`, since `numberOfDoors` is an instance variable in the class *Car*. But since class *Car* extends class *Vehicle*, a car also has all the structure and behavior of a vehicle. This means that `myCar.registrationNumber`, `myCar.owner`, and `myCar.transferOwnership()` also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type *Car* or *Truck* or *Motorcycle* is automatically an object of type *Vehicle* too. This brings us to the following Important Fact:

**A variable that can hold a reference
to an object of class A can also hold a reference
to an object belonging to any subclass of A.**

The practical effect of this in our example is that an object of type *Car* can be assigned to a variable of type *Vehicle*. That is, it would be legal to say

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

After either of these statements, the variable `myVehicle` holds a reference to a *Vehicle* object that happens to be an instance of the subclass, *Car*. The object “remembers” that it is in fact a *Car*, and not **just** a *Vehicle*. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the `instanceof` operator. The test:

```
if (myVehicle instanceof Car) ...
```

determines whether the object referred to by `myVehicle` is in fact a car.

On the other hand, the assignment statement

```
myCar = myVehicle;
```

would be illegal because `myVehicle` could potentially refer to other types of vehicles that are not cars. This is similar to a problem we saw previously in Subsection 2.5.6: The computer will not allow you to assign an **int** value to a variable of type **short**, because not every **int** is a **short**. Similarly, it will not allow you to assign a value of type *Vehicle* to a variable of type *Car* because not every vehicle is a car. As in the case of **ints** and **shorts**, the solution here is to use type-casting. If, for some reason, you happen to know that `myVehicle` does in fact refer to a *Car*, you can use the type cast `(Car)myVehicle` to tell the computer to treat `myVehicle` as if it were actually of type *Car*. So, you could say

```
myCar = (Car)myVehicle;
```

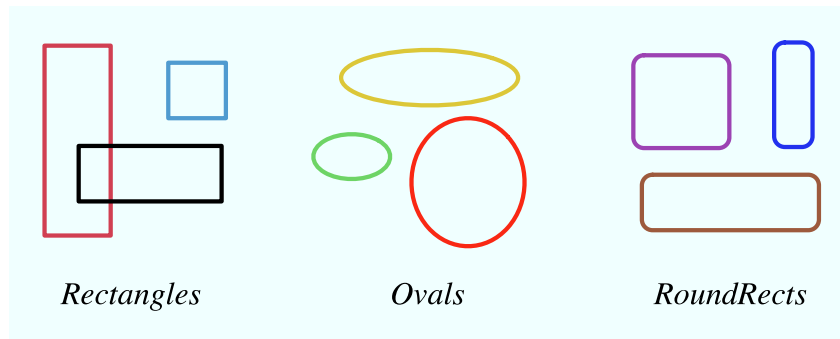
and you could even refer to `((Car)myVehicle).numberOfDoors`. As an example of how this could be used in a program, suppose that you want to print out relevant data about a vehicle. You could say:

```
System.out.println("Vehicle Data:");
System.out.println("Registration number: "
                  + myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
    System.out.println("Type of vehicle: Car");
    Car c;
    c = (Car)myVehicle;
    System.out.println("Number of doors: " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
    System.out.println("Type of vehicle: Truck");
    Truck t;
    t = (Truck)myVehicle;
    System.out.println("Number of axles: " + t.numberOfAxles);
}
else if (myVehicle instanceof Motorcycle) {
    System.out.println("Type of vehicle: Motorcycle");
    Motorcycle m;
    m = (Motorcycle)myVehicle;
    System.out.println("Has a sidecar: " + m.hasSidecar);
}
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if `myVehicle` refers to an object of type *Truck*, then the type cast `(Car)myVehicle` would be an error. When this happens, an exception of type *ClassCastException* is thrown.

5.5.4 Polymorphism

As another example, consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors. (A “roundrect” is just a rectangle with rounded corners.)



Three classes, *Rectangle*, *Oval*, and *RoundRect*, could be used to represent the three types of shapes. These three classes would have a common superclass, *Shape*, to represent features that all three shapes have in common. The *Shape* class could include instance variables to represent the color, position, and size of a shape, and it could include instance methods for changing the color, position, and size. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {
    Color color;    // Color of the shape. (Recall that class Color
                  // is defined in package java.awt. Assume
                  // that this class has been imported.)

    void setColor(Color newColor) {
        // Method to change the color of the shape.
        color = newColor; // change value of instance variable
        redraw(); // redraw shape, which will appear in new color
    }

    void redraw() {
        // method for drawing the shape
        ??? // what commands should go here?
    }

    . . .          // more instance variables and methods
} // end of class Shape
```

Now, you might see a problem here with the method `redraw()`. The problem is that each different type of shape is drawn differently. The method `setColor()` can be called for any type of shape. How does the computer know which shape to draw when it executes the `redraw()`? Informally, we can answer the question like this: The computer executes `redraw()` by asking the shape to redraw **itself**. Every shape object knows what it has to do to redraw itself.

In practice, this means that each of the specific shape classes has its own `redraw()` method:

```
class Rectangle extends Shape {
    void redraw() {
        . . . // commands for drawing a rectangle
    }
}
```

```

    }
    . . . // possibly, more methods and variables
}
class Oval extends Shape {
    void redraw() {
        . . . // commands for drawing an oval
    }
    . . . // possibly, more methods and variables
}
class RoundRect extends Shape {
    void redraw() {
        . . . // commands for drawing a rounded rectangle
    }
    . . . // possibly, more methods and variables
}

```

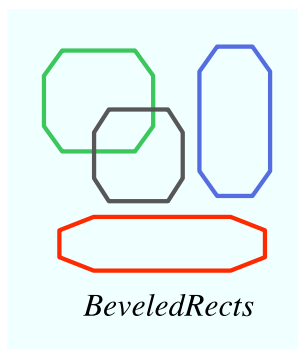
If `oneShape` is a variable of type *Shape*, it could refer to an object of any of the types, *Rectangle*, *Oval*, or *RoundRect*. As a program executes, and the value of `oneShape` changes, it could even refer to objects of different types at different times! Whenever the statement

```
oneShape.redraw();
```

is executed, the redraw method that is actually called is the one appropriate for the type of object to which `oneShape` actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that `oneShape` happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of `oneShape` changes as the loop is executed, it is possible that the very same statement “`oneShape.redraw();`” will call different methods and draw different shapes as it is executed over and over. We say that the `redraw()` method is *polymorphic*. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming.

Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a *message* to an object. The object responds to the message by executing the appropriate method. The statement “`oneShape.redraw();`” is a message to the object referred to by `oneShape`. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes “`oneShape.redraw();`” in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages, and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.

One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn’t even conceive of, at the time you wrote it. Suppose that I decide to add beveled rectangles to the types of shapes my program can deal with. A beveled rectangle has a triangle cut off each corner:



To implement beveled rectangles, I can write a new subclass, *BeveledRect*, of class *Shape* and give it its own `redraw()` method. Automatically, code that I wrote previously—such as the statement `oneShape.redraw()`—can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn’t exist when I wrote the statement!

In the statement “`oneShape.redraw()`;”, the `redraw` message is sent to the object `oneShape`. Look back at the method from the *Shape* class for changing the color of a shape:

```
void setColor(Color newColor) {
    color = newColor; // change value of instance variable
    redraw(); // redraw shape, which will appear in new color
}
```

A `redraw` message is sent here, but which object is it sent to? Well, the `setColor` method is itself a message that was sent to some object. The answer is that the `redraw` message is sent to that same object, the one that received the `setColor` message. If that object is a rectangle, then it is the `redraw()` method from the *Rectangle* class that is executed. If the object is an oval, then it is the `redraw()` method from the *Oval* class. This is what you should expect, but it means that the “`redraw()`;” statement in the `setColor()` method does **not** necessarily call the `redraw()` method in the *Shape* class! The `redraw()` method that is executed could be in any subclass of *Shape*.

Again, this is not a real surprise if you think about it in the right way. Remember that an instance method is always contained in an object. The class only contains the source code for the method. When a *Rectangle* object is created, it contains a `redraw()` method. The source code for that method is in the *Rectangle* class. The object also contains a `setColor()` method. Since the *Rectangle* class does not define a `setColor()` method, the **source code** for the rectangle’s `setColor()` method comes from the superclass, *Shape*, but the **method itself** is in the object of type *Rectangle*. Even though the source codes for the two methods are in different classes, the methods themselves are part of the same object. When the rectangle’s `setColor()` method is executed and calls `redraw()`, the `redraw()` method that is executed is the one in the same object.

5.5.5 Abstract Classes

Whenever a *Rectangle*, *Oval*, or *RoundRect* object has to draw itself, it is the `redraw()` method in the appropriate class that is executed. This leaves open the question, What does the `redraw()` method in the *Shape* class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class *Shape* represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there

even be a `redraw()` method in the *Shape* class? Well, it has to be there, or it would be illegal to call it in the `setColor()` method of the *Shape* class, and it would be illegal to write “`oneShape.redraw();`”, where `oneShape` is a variable of type *Shape*. The compiler would complain that `oneShape` is a variable of type *Shape* and there’s no `redraw()` method in the *Shape* class.

Nevertheless the version of `redraw()` in the *Shape* class itself will never actually be called. In fact, if you think about it, there can never be any reason to construct an actual object of type *Shape*! You can have **variables** of type *Shape*, but the objects they refer to will always belong to one of the subclasses of *Shape*. We say that *Shape* is an **abstract class**. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses. A class that is not abstract is said to be **concrete**. You can create objects belonging to a concrete class, but not to an abstract class. A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.

Similarly, we say that the `redraw()` method in class *Shape* is an **abstract method**, since it is never meant to be called. In fact, there is nothing for it to do—any actual redrawing is done by `redraw()` methods in the subclasses of *Shape*. The `redraw()` method in *Shape* has to be there. But it is there only to tell the computer that all *Shapes* understand the `redraw` message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of `redraw()` in the subclasses of *Shape*. There is no reason for the abstract `redraw()` in class *Shape* to contain any code at all.

Shape and its `redraw()` method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier “**abstract**” to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must be provided for the abstract method in any concrete subclass of the abstract class. Here’s what the *Shape* class would look like as an abstract class:

```
public abstract class Shape {
    Color color;    // color of shape.

    void setColor(Color newColor) {
        // method to change the color of the shape
        color = newColor; // change value of instance variable
        redraw(); // redraw shape, which will appear in new color
    }

    abstract void redraw();
        // abstract method---must be defined in
        // concrete subclasses

    . . .        // more instance variables and methods
} // end of class Shape
```

Once you have declared the class to be **abstract**, it becomes illegal to try to create actual objects of type *Shape*, and the computer will report a syntax error if you try to do so.

* * *

Recall from Subsection 5.3.3 that a class that is not explicitly declared to be a subclass of some other class is automatically made a subclass of the standard class *Object*. That is, a class declaration with no “**extends**” part such as

```
public class myClass { . . .
```

is exactly equivalent to

```
public class myClass extends Object { . . .
```

This means that class *Object* is at the top of a huge class hierarchy that includes every other class. (Semantically, *Object* is an abstract class, in fact the most abstract class of all. Curiously, however, it is not declared to be **abstract** syntactically, which means that you can create objects of type *Object*. What you would do with them, however, I have no idea.)

Since every class is a subclass of *Object*, a variable of type *Object* can refer to any object whatsoever, of any type. Java has several standard data structures that are designed to hold *Objects*, but since every object is an instance of class *Object*, these data structures can actually hold any object whatsoever. One example is the “*ArrayList*” data structure, which is defined by the class *ArrayList* in the package `java.util`. (*ArrayList* is discussed more fully in Section 7.3.) An *ArrayList* is simply a list of *Objects*. This class is very convenient, because an *ArrayList* can hold any number of objects, and it will grow, when necessary, as objects are added to it. Since the items in the list are of type *Object*, the list can actually hold objects of any type.

A program that wants to keep track of various *Shapes* that have been drawn on the screen can store those shapes in an *ArrayList*. Suppose that the *ArrayList* is named `listOfShapes`. A shape, `oneShape`, can be added to the end of the list by calling the instance method “`listOfShapes.add(oneShape);`”. The shape can be removed from the list with the instance method “`listOfShapes.remove(oneShape);`”. The number of shapes in the list is given by the function “`listOfShapes.size()`”. And it is possible to retrieve the *i*-th object from the list with the function call “`listOfShapes.get(i)`”. (Items in the list are numbered from 0 to `listOfShapes.size() - 1`.) However, note that this method returns an *Object*, not a *Shape*. (Of course, the people who wrote the *ArrayList* class didn’t even know about *Shapes*, so the method they wrote could hardly have a return type of *Shape*!) Since you know that the items in the list are, in fact, *Shapes* and not just *Objects*, you can type-cast the *Object* returned by `listOfShapes.get(i)` to be a value of type *Shape*:

```
oneShape = (Shape)listOfShapes.get(i);
```

Let’s say, for example, that you want to redraw all the shapes in the list. You could do this with a simple `for` loop, which is lovely example of object-oriented programming and of polymorphism:

```
for (int i = 0; i < listOfShapes.size(); i++) {
    Shape s; // i-th element of the list, considered as a Shape
    s = (Shape)listOfShapes.get(i);
    s.redraw(); // What is drawn here depends on what type of shape s is!
}
```

* * *

The sample source code file *ShapeDraw.java* uses an abstract *Shape* class and an *ArrayList* to hold a list of shapes. The file defines an applet in which the user can add various shapes to a drawing area. Once a shape is in the drawing area, the user can use the mouse to drag it around.

You might want to look at this file, even though you won’t be able to understand all of it at this time. Even the definitions of the shape classes are somewhat different from those that I have described in this section. (For example, the `draw()` method has a parameter of type *Graphics*. This parameter is required because of the way Java handles all drawing.) I’ll return

to this example in later chapters when you know more about GUI programming. However, it would still be worthwhile to look at the definition of the *Shape* class and its subclasses in the source code. You might also check how an `ArrayList` is used to hold the list of shapes.

In the applet the only time when the actual class of a shape is used is when that shape is added to the screen. Once the shape has been created, it is manipulated entirely as an abstract shape. The routine that implements dragging, for example, works only with variables of type *Shape*. As the *Shape* is being dragged, the dragging routine just calls the *Shape*'s draw method each time the shape has to be drawn, so it doesn't have to know how to draw the shape or even what type of shape it is. The object is responsible for drawing itself. If I wanted to add a new type of shape to the program, I would define a new subclass of *Shape*, add another button to the applet, and program the button to add the correct type of shape to the screen. No other changes in the programming would be necessary.

If you want to try out the applet, you can find it at the end of the on-line version of this section.

5.6 this and super

ALTHOUGH THE BASIC IDEAS of object-oriented programming are reasonably simple and clear, they are subtle, and they take time to get used to. And unfortunately, beyond the basic ideas there are a lot of details. This section and the next cover more of those annoying details. You should not necessarily master everything in these two sections the first time through, but you should read it to be aware of what is possible. For the most part, when I need to use this material later in the text, I will explain it again briefly, or I will refer you back to it. In this section, we'll look at two variables, `this` and `super`, that are automatically defined in any instance method.

5.6.1 The Special Variable `this`

A static member of a class has a simple name, which can only be used inside the class definition. For use outside the class, it has a full name of the form `<class-name>.<simple-name>`. For example, "`System.out`" is a static member variable with simple name "out" in the class "`System`". It's always legal to use the full name of a static member, even within the class where it's defined. Sometimes it's even necessary, as when the simple name of a static member variable is hidden by a local variable of the same name.

Instance variables and instance methods also have simple names. The simple name of such an instance member can be used in instance methods in the class where the instance member is defined. Instance members also have full names, but remember that instance variables and methods are actually contained in objects, not classes. The full name of an instance member has to contain a reference to the object that contains the instance member. To get at an instance variable or method from outside the class definition, you need a variable that refers to the object. Then the full name is of the form `<variable-name>.<simple-name>`. But suppose you are writing the definition of an instance method in some class. How can you get a reference to the object that contains that instance method? You might need such a reference, for example, if you want to use the full name of an instance variable, because the simple name of the instance variable is hidden by a local variable or parameter.

Java provides a special, predefined variable named "`this`" that you can use for such purposes. The variable, `this`, is used in the source code of an instance method to refer to the

object that contains the method. This intent of the name, `this`, is to refer to “this object,” the one right here that this very method is in. If `x` is an instance variable in the same object, then `this.x` can be used as a full name for that variable. If `otherMethod()` is an instance method in the same object, then `this.otherMethod()` could be used to call that method. Whenever the computer executes an instance method, it automatically sets the variable, `this`, to refer to the object that contains the method.

One common use of `this` is in constructors. For example:

```
public class Student {
    private String name; // Name of the student.

    public Student(String name) {
        // Constructor. Create a student with specified name.
        this.name = name;
    }
    .
    . // More variables and methods.
    .
}
```

In the constructor, the instance variable called `name` is hidden by a formal parameter. However, the instance variable can still be referred to by its full name, `this.name`. In the assignment statement, the value of the formal parameter, `name`, is assigned to the instance variable, `this.name`. This is considered to be acceptable style: There is no need to dream up cute new names for formal parameters that are just used to initialize instance variables. You can use the same name for the parameter as for the instance variable.

There are other uses for `this`. Sometimes, when you are writing an instance method, you need to pass the object that contains the method to a subroutine, as an actual parameter. In that case, you can use `this` as the actual parameter. For example, if you wanted to print out a string representation of the object, you could say “`System.out.println(this);`”. Or you could assign the value of `this` to another variable in an assignment statement. In fact, you can do anything with `this` that you could do with any other variable, except change its value.

5.6.2 The Special Variable `super`

Java also defines another special variable, named “`super`”, for use in the definitions of instance methods. The variable `super` is for use in a subclass. Like `this`, `super` refers to the object that contains the method. But it’s forgetful. It forgets that the object belongs to the class you are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. `super` doesn’t know about any of those additions and modifications; it can only be used to refer to methods and variables in the superclass.

Let’s say that the class that you are writing contains an instance method named `doSomething()`. Consider the subroutine call statement `super.doSomething()`. Now, `super` doesn’t know anything about the `doSomething()` method in the subclass. It only knows about things in the superclass, so it tries to execute a method named `doSomething()` from the superclass. If there is none—if the `doSomething()` method was an addition rather than a modification—you’ll get a syntax error.

The reason `super` exists is so you can get access to things in the superclass that are **hidden** by things in the subclass. For example, `super.x` always refers to an instance variable named

`x` in the superclass. This can be useful for the following reason: If a class contains an instance variable with the same name as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass. The variable in the subclass does not **replace** the variable of the same name in the superclass; it merely *hides* it. The variable from the superclass can still be accessed, using `super`.

When you write a method in a subclass that has the same signature as a method in its superclass, the method from the superclass is hidden in the same way. We say that the method in the subclass **overrides** the method from the superclass. Again, however, `super` can be used to access the method from the superclass.

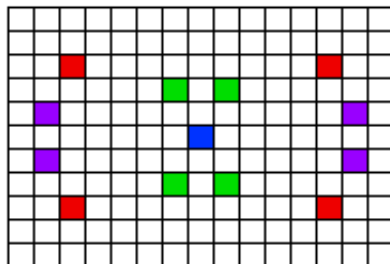
The major use of `super` is to override a method with a new method that **extends** the behavior of the inherited method, instead of **replacing** that behavior entirely. The new method can use `super` to call the method from the superclass, and then it can add additional code to provide additional behavior. As an example, suppose you have a *PairOfDice* class that includes a `roll()` method. Suppose that you want a subclass, *GraphicalDice*, to represent a pair of dice drawn on the computer screen. The `roll()` method in the *GraphicalDice* class should do everything that the `roll()` method in the *PairOfDice* class does. We can express this with a call to `super.roll()`, which calls the method in the superclass. But in addition to that, the `roll()` method for a *GraphicalDice* object has to redraw the dice to show the new values. The *GraphicalDice* class might look something like this:

```
public class GraphicalDice extends PairOfDice {
    public void roll() {
        // Roll the dice, and redraw them.
        super.roll(); // Call the roll method from PairOfDice.
        redraw();    // Call a method to draw the dice.
    }
    .
    . // More stuff, including definition of redraw().
    .
}
```

Note that this allows you to extend the behavior of the `roll()` method even if you don't know how the method is implemented in the superclass!

Here is a more complete example. The applet at the end of Section 4.7 in the on-line version of this book shows a disturbance that moves around in a mosaic of little squares. As it moves, each square that it visits becomes a brighter shade of red. The result looks interesting, but I think it would be prettier if the pattern were symmetric. A symmetric version of the applet is shown at the bottom of Section 5.7 (in the on-line version). The symmetric applet can be programmed as an easy extension of the original applet.

In the symmetric version, each time a square is brightened, the squares that can be obtained from that one by horizontal and vertical reflection through the center of the mosaic are also brightened. This picture might make the symmetry idea clearer:



The four red squares in the picture, for example, form a set of such symmetrically placed squares, as do the purple squares and the green squares. (The blue square is at the center of the mosaic, so reflecting it doesn't produce any other squares; it's its own reflection.)

The original applet is defined by the class *RandomBrighten*. In that class, the actual task of brightening a square is done by a method called `brighten()`. If `row` and `col` are the row and column numbers of a square, then "`brighten(row,col);`" increases the brightness of that square. All we need is a subclass of *RandomBrighten* with a modified `brighten()` routine. Instead of just brightening one square, the modified routine will also brighten the horizontal and vertical reflections of that square. But how will it brighten each of the four individual squares? By calling the `brighten()` method from the original class. It can do this by calling `super.brighten()`.

There is still the problem of computing the row and column numbers of the horizontal and vertical reflections. To do this, you need to know the number of rows and the number of columns. The *RandomBrighten* class has instance variables named `ROWS` and `COLUMNS` to represent these quantities. Using these variables, it's possible to come up with formulas for the reflections, as shown in the definition of the `brighten()` method below.

Here's the complete definition of the new class:

```
public class SymmetricBrighten extends RandomBrighten {
    void brighten(int row, int col) {
        // Brighten the specified square and its horizontal
        // and vertical reflections. This overrides the brighten
        // method from the RandomBrighten class, which just
        // brightens one square.
        super.brighten(row, col);
        super.brighten(ROWS - 1 - row, col);
        super.brighten(row, COLUMNS - 1 - col);
        super.brighten(ROWS - 1 - row, COLUMNS - 1 - col);
    }
} // end class SymmetricBrighten
```

This is the entire source code for the applet!

5.6.3 Constructors in Subclasses

Constructors are not inherited. That is, if you extend an existing class to make a subclass, the constructors in the superclass do not become part of the subclass. If you want constructors in the subclass, you have to define new ones from scratch. If you don't define any constructors in the subclass, then the computer will make up a default constructor, with no parameters, for you.

This could be a problem, if there is a constructor in the superclass that does a lot of necessary work. It looks like you might have to repeat all that work in the subclass! This could be a **real** problem if you don't have the source code to the superclass, and don't know how it works, or if the constructor in the superclass initializes **private** member variables that you don't even have access to in the subclass!

Obviously, there has to be some fix for this, and there is. It involves the special variable, **super**. As the very first statement in a constructor, you can use **super** to call a constructor from the superclass. The notation for this is a bit ugly and misleading, and it can only be used in this one particular circumstance: It looks like you are calling **super** as a subroutine (even though **super** is not a subroutine and you can't call constructors the same way you call other subroutines anyway). As an example, assume that the *PairOfDice* class has a constructor that takes two integers as parameters. Consider a subclass:

```
public class GraphicalDice extends PairOfDice {

    public GraphicalDice() { // Constructor for this class.

        super(3,4); // Call the constructor from the
                   // PairOfDice class, with parameters 3, 4.

        initializeGraphics(); // Do some initialization specific
                               // to the GraphicalDice class.
    }

    .
    . // More constructors, methods, variables...
    .
}
```

The statement “**super(3,4);**” calls the constructor from the superclass. This call must be the first line of the constructor in the subclass. Note that if you don't explicitly call a constructor from the superclass in this way, then the default constructor from the superclass, the one with no parameters, will be called automatically.

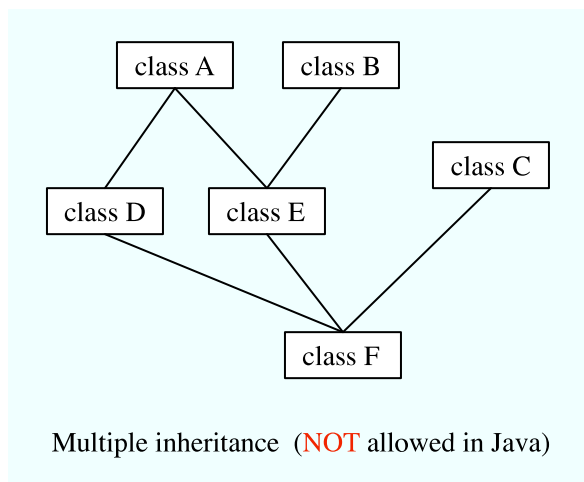
This might seem rather technical, but unfortunately it is sometimes necessary. By the way, you can use the special variable **this** in exactly the same way to call another constructor in the same class. This can be useful since it can save you from repeating the same code in several constructors.

5.7 Interfaces, Nested Classes, and Other Details

THIS SECTION simply pulls together a few more miscellaneous features of object oriented programming in Java. Read it now, or just look through it and refer back to it later when you need this material. (You will need to know about the first topic, interfaces, almost as soon as we begin GUI programming.)

5.7.1 Interfaces

Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses. This is called **multiple inheritance**. In the illustration below, for example, class E is shown as having both class A and class B as direct superclasses, while class F has three direct superclasses.



Such multiple inheritance is **not** allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritance were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: *interfaces*.

We’ve encountered the term “interface” before, in connection with black boxes in general and subroutines in particular. The interface of a subroutine consists of the name of the subroutine, its return type, and the number and types of its parameters. This is the information you need to know if you want to call the subroutine. A subroutine also has an implementation: the block of code which defines it and which is executed when the subroutine is called.

In Java, **interface** is a reserved word with an additional, technical meaning. An “**interface**” in this sense consists of a set of instance method interfaces, without any associated implementations. (Actually, a Java interface can contain other things as well, but we won’t discuss them here.) A class can *implement* an **interface** by providing an implementation for each of the methods specified by the interface. Here is an example of a very simple Java **interface**:

```

public interface Drawable {
    public void draw(Graphics g);
}

```

This looks much like a class definition, except that the implementation of the **draw()** method is omitted. A class that implements the **interface** *Drawable* must provide an implementation for this method. Of course, the class can also include other methods and variables. For example,

```

public class Line implements Drawable {
    public void draw(Graphics g) {
        . . . // do something---presumably, draw a line
    }
    . . . // other methods and variables
}

```

Note that to implement an interface, a class must do more than simply provide an implementation for each method in the interface; it must also **state** that it implements the interface, using the reserved word **implements** as in this example: “public class Line **implements** Drawable”. Any class that implements the *Drawable* interface defines a **draw()** instance method. Any object created from such a class includes a **draw()** method. We say that an **object** implements

an **interface** if it belongs to a class that implements the interface. For example, any object of type *Line* implements the *Drawable* interface.

While a class can **extend** only one other class, it can **implement** any number of interfaces. In fact, a class can both extend one other class and implement one or more interfaces. So, we can have things like

```
class FilledCircle extends Circle
                                implements Drawable, Fillable {
    . . .
}
```

The point of all this is that, although interfaces are not classes, they are something very similar. An interface is very much like an abstract class, that is, a class that can never be used for constructing objects, but can be used as a basis for making subclasses. The subroutines in an interface are abstract methods, which must be implemented in any concrete class that implements the interface. And as with abstract classes, even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if *Drawable* is an interface, and if *Line* and *FilledCircle* are classes that implement *Drawable*, then you could say:

```
Drawable figure; // Declare a variable of type Drawable. It can
                  // refer to any object that implements the
                  // Drawable interface.

figure = new Line(); // figure now refers to an object of class Line
figure.draw(g);      // calls draw() method from class Line

figure = new FilledCircle(); // Now, figure refers to an object
                             // of class FilledCircle.
figure.draw(g);          // calls draw() method from class FilledCircle
```

A variable of type *Drawable* can refer to any object of any class that implements the *Drawable* interface. A statement like `figure.draw(g)`, above, is legal because `figure` is of type *Drawable*, and **any** *Drawable* object has a `draw()` method. So, whatever object `figure` refers to, that object must have a `draw()` method.

Note that a **type** is something that can be used to declare variables. A type can also be used to specify the type of a parameter in a subroutine, or the return type of a function. In Java, a type can be either a class, an interface, or one of the eight built-in primitive types. These are the only possibilities. Of these, however, only classes can be used to construct new objects.

You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are a few interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few chapters.

5.7.2 Nested Classes

A class seems like it should be a pretty important thing. A class is a high-level building block of a program, representing a potentially complex idea and its associated data and behaviors. I've always felt a bit silly writing tiny little classes that exist only to group a few scraps of data together. However, such trivial classes are often useful and even essential. Fortunately, in Java, I can ease the embarrassment, because one class can be nested inside another class. My trivial

little class doesn't have to stand on its own. It becomes part of a larger more respectable class. This is particularly useful when you want to create a little class specifically to support the work of a larger class. And, more seriously, there are other good reasons for nesting the definition of one class inside another class.

In Java, a *nested class* is any class whose definition is inside the definition of another class. Nested classes can be either *named* or *anonymous*. I will come back to the topic of anonymous classes later in this section. A named nested class, like most other things that occur in classes, can be either static or non-static.

The definition of a static nested class looks just like the definition of any other class, except that it is nested inside another class and it has the modifier `static` as part of its declaration. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it has not been declared private, then it can also be used outside the containing class, but when it is used outside the class, its name must indicate its membership in the containing class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named *WireFrameModel* represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the *WireFrameModel* class contains a static nested class, *Line*, that represents a single line. Then, outside of the class *WireFrameModel*, the *Line* class would be referred to as `WireFrameModel.Line`. Of course, this just follows the normal naming convention for static members of a class. The definition of the *WireFrameModel* class with its nested *Line* class would look, in outline, like this:

```
public class WireFrameModel {
    . . . // other members of the WireFrameModel class

    static public class Line {
        // Represents a line from the point (x1,y1,z1)
        // to the point (x2,y2,z2) in 3-dimensional space.
        double x1, y1, z1;
        double x2, y2, z2;
    } // end class Line

    . . . // other members of the WireFrameModel class
} // end WireFrameModel
```

Inside the *WireFrameModel* class, a *Line* object would be created with the constructor “`new Line()`”. Outside the class, “`new WireFrameModel.Line()`” would be used.

A static nested class has full access to the static members of the containing class, even to the private members. Similarly, the containing class has full access to the members of the nested class. This can be another motivation for declaring a nested class, since it lets you give one class access to the private members of another class without making those members generally available to other classes.

When you compile the above class definition, two class files will be created. Even though the definition of *Line* is nested inside *WireFrameModel*, the compiled *Line* class is stored in a separate file. The name of the class file for *Line* will be `WireFrameModel$Line.class`.

* * *

Non-static nested classes are referred to as *inner classes*. Inner classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated with an object rather than to the class in which it is nested. This can take some getting used to.

Any non-static member of a class is not really part of the class itself (although its source code is contained in the class definition). This is true for inner classes, just as it is for any other non-static part of a class. The non-static members of a class specify what will be contained in objects that are created from that class. The same is true—at least logically—for inner classes. It’s as if each object that belongs to the containing class has its **own copy** of the nested class. This copy has access to all the instance methods and instance variables of the object, even to those that are declared **private**. The two copies of the inner class in two different objects differ because the instance variables and methods they refer to are in different objects. In fact, the rule for deciding whether a nested class should be static or non-static is simple: If the nested class needs to use any instance variable or instance method from the containing class, make the nested class non-static. Otherwise, it might as well be static.

From outside the containing class, a non-static nested class has to be referred to using a name of the form $\langle variableName \rangle . \langle NestedClassName \rangle$, where $\langle variableName \rangle$ is a variable that refers to the object that contains the class. This is actually rather rare, however. A non-static nested class is generally used only inside the class in which it is nested, and there it can be referred to by its simple name.

In order to create an object that belongs to an inner class, you must first have an object that belongs to the containing class. (When working inside the class, the object “**this**” is used implicitly.) The inner class object is permanently associated with the containing class object, and it has complete access to the members of the containing class object. Looking at an example will help, and will hopefully convince you that inner classes are really very natural. Consider a class that represents poker games. This class might include a nested class to represent the players of the game. This structure of the *PokerGame* class could be:

```
public class PokerGame { // Represents a game of poker.

    private class Player { // Represents one of the players in this game.
        .
        .
        .
    } // end class Player

    private Deck deck; // A deck of cards for playing the game.
    private int pot; // The amount of money that has been bet.

    .
    .
    .

} // end class PokerGame
```

If *game* is a variable of type *PokerGame*, then, conceptually, *game* contains its own copy of the *Player* class. In an instance method of a *PokerGame* object, a new *Player* object would be created by saying “**new Player()**”, just as for any other class. (A *Player* object could be created outside the *PokerGame* class with an expression such as “*game.new Player()*”. Again, however, this is very rare.) The *Player* object will have access to the **deck** and **pot** instance variables in the *PokerGame* object. Each *PokerGame* object has its own **deck** and **pot** and **Players**. Players of that poker game use the deck and pot for that game; players of another poker game use the other game’s deck and pot. That’s the effect of making the *Player* class

non-static. This is the most natural way for players to behave. A *Player* object represents a player of one particular poker game. If *Player* were a **static** nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

5.7.3 Anonymous Inner Classes

In some cases, you might find yourself writing an inner class and then using that class in just a single line of your program. Is it worth creating such a class? Indeed, it can be, but for cases like this you have the option of using an *anonymous inner class*. An anonymous class is created with a variation of the **new** operator that has the form

```
new <superclass-or-interface> ( <parameter-list> ) {
    <methods-and-variables>
}
```

This constructor defines a new class, without giving it a name, and it simultaneously creates an object that belongs to that class. This form of the **new** operator can be used in any statement where a regular “**new**” could be used. The intention of this expression is to create: “a new object belonging to a class that is the same as *<superclass-or-interface>* but with these *<methods-and-variables>* added.” The effect is to create a uniquely customized object, just at the point in the program where you need it. Note that it is possible to base an anonymous class on an interface, rather than a class. In this case, the anonymous class must implement the interface by defining all the methods that are declared in the interface. If an interface is used as a base, the *<parameter-list>* must be empty. Otherwise, it can contain parameters for a constructor in the *<superclass>*.

Anonymous classes are often used for handling events in graphical user interfaces, and we will encounter them several times in the chapters on GUI programming. For now, we will look at one not-very-plausible example. Consider the *Drawable* interface, which is defined earlier in this section. Suppose that we want a *Drawable* object that draws a filled, red, 100-pixel square. Rather than defining a new, separate class and then using that class to create the object, we can use an anonymous class to create the object in one statement:

```
Drawable redSquare = new Drawable() {
    void draw(Graphics g) {
        g.setColor(Color.red);
        g.fillRect(10,10,100,100);
    }
};
```

The semicolon at the end of this statement is not part of the class definition. It’s the semicolon that is required at the end of every declaration statement.

When a Java class is compiled, each anonymous nested class will produce a separate class file. If the name of the main class is *MainClass*, for example, then the names of the class files for the anonymous nested classes will be *MainClass\$1.class*, *MainClass\$2.class*, *MainClass\$3.class*, and so on.

5.7.4 Mixing Static and Non-static

Classes, as I’ve said, have two very distinct purposes. A class can be used to group together a set of static member variables and static member subroutines. Or it can be used as a factory for making objects. The non-static variables and subroutines in the class definition specify the

instance variables and methods of the objects. In most cases, a class performs one or the other of these roles, not both.

Sometimes, however, static and non-static members are mixed in a single class. In this case, the class plays a dual role. Sometimes, these roles are completely separate. It is also possible for the static and non-static parts of a class to interact. This happens when instance methods use static member variables or call static member subroutines. An instance method belongs to an object, not to the class itself, and there can be many objects with their own versions of the instance method. But there is only one copy of a static member variable. So, effectively, we have many objects sharing that one variable.

Suppose, for example, that we want to write a *PairOfDice* class that uses the *Random* class mentioned in Section 5.3 for rolling the dice. To do this, a *PairOfDice* object needs access to an object of type *Random*. But there is no need for each *PairOfDice* object to have a separate *Random* object. (In fact, it would not even be a good idea: Because of the way random number generators work, a program should, in general, use only one source of random numbers.) A nice solution is to have a single *Random* variable as a static member of the *PairOfDice* class, so that it can be shared by all *PairOfDice* objects. For example:

```
import java.util.Random;

public class PairOfDice {

    private static Random randGen = new Random();

    public int die1;    // Number showing on the first die.
    public int die2;    // Number showing on the second die.

    public PairOfDice() {
        // Constructor. Creates a pair of dice that
        // initially shows random values.
        roll();
    }

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = randGen.nextInt(6) + 1;
        die2 = randGen.nextInt(6) + 1;
    }

} // end class PairOfDice
```

As another example, let's rewrite the *Student* class that was used in Section 5.2. I've added an ID for each student and a static member called `nextUniqueID`. Although there is an ID variable in each student object, there is only one `nextUniqueID` variable.

```
public class Student {

    private String name; // Student's name.
    private int ID; // Unique ID number for this student.
    public double test1, test2, test3; // Grades on three tests.

    private static int nextUniqueID = 0;
        // keep track of next available unique ID number

    Student(String theName) {
        // Constructor for Student objects; provides a name for the Student,
```

```

        // and assigns the student a unique ID number.
        name = theName;
        nextUniqueID++;
        ID = nextUniqueID;
    }

    public String getName() {
        // Accessor method for reading value of private
        // instance variable, name.
        return name;
    }

    public int getID() {
        // Accessor method for reading value of ID.
        return ID;
    }

    public double getAverage() {
        // Compute average test grade.
        return (test1 + test2 + test3) / 3;
    }
} // end of class Student

```

The initialization “`nextUniqueID = 0`” is done only once, when the class is first loaded. Whenever a *Student* object is constructed and the constructor says “`nextUniqueID++`”, it’s always the same static member variable that is being incremented. When the very first *Student* object is created, `nextUniqueID` becomes 1. When the second object is created, `nextUniqueID` becomes 2. After the third object, it becomes 3. And so on. The constructor stores the new value of `nextUniqueID` in the `ID` variable of the object that is being created. Of course, `ID` is an instance variable, so every object has its own individual `ID` variable. The class is constructed so that each student will automatically get a different value for its `ID` variable. Furthermore, the `ID` variable is *private*, so there is no way for this variable to be tampered with after the object has been created. You are guaranteed, just by the way the class is designed, that every student object will have its own permanent, unique identification number. Which is kind of cool if you think about it.

5.7.5 Static Import

The `import` directive makes it possible to refer to a class such as `java.awt.Color` using its simple name, *Color*. All you have to do is say `import java.awt.Color` or `import java.awt.*`. But you still have to use compound names to refer to static member variables such as `System.out` and to static methods such as `Math.sqrt`.

Java 5.0 introduced a new form of the `import` directive that can be used to import *static* members of a class in the same way that the ordinary `import` directive imports classes from a package. The new form of the directive is called a *static import*, and it has syntax

```
import static <package-name>.<class-name>.<static-member-name>;
```

to import one static member name from a class, or

```
import static <package-name>.<class-name>.*;
```

to import all the public static members from a class. For example, if you preface a class definition with

```
import static java.lang.System.out;
```

then you can use the simple name `out` instead of the compound name `System.out`. This means you can use `out.println` instead of `System.out.println`. If you are going to work extensively with the *Math* class, you can preface your class definition with

```
import static java.lang.Math.*;
```

This would allow you to say `sqrt` instead of `Math.sqrt`, `log` instead of `Math.log`, `PI` instead of `Math.PI`, and so on.

Note that the static import directive requires a *<package-name>*, even for classes in the standard package `java.lang`. One consequence of this is that you can't do a static import from a class in the default package. In particular, it is not possible to do a static import from my *TextIO* class—if you wanted to do that, you would have to move *TextIO* into a package.

5.7.6 Enums as Classes

Enumerated types were introduced in Subsection 2.3.3. Now that we have covered more material on classes and objects, we can revisit the topic (although still not covering enumerated types in their full complexity).

Enumerated types are actually classes, and each enumerated type constant is a **public**, **final**, **static** member variable in that class (even though they are not declared with these modifiers). The value of the variable is an object belonging to the enumerated type class. There is one such object for each enumerated type constant, and these are the only objects of the class that can ever be created. It is really these objects that represent the possible values of the enumerated type. The enumerated type constants are actually variables that refer to these objects.

When an enumerated type is defined inside another class, it is a nested class inside the enclosing class. In fact, it is a static nested class, whether you declare it to be **static** or not. But it can also be declared as a non-nested class, in a file of its own. For example, we could define the following enumerated type in a file named *Suit.java*:

```
public enum Suit {
    SPADES, HEARTS, DIAMONDS, CLUBS
}
```

This enumerated type represents the four possible suits for a playing card, and it could have been used in the example *Card.java* from Subsection 5.4.2.

Furthermore, in addition to its list of values, an enumerated type can contain some of the other things that a regular class can contain, including methods and additional member variables. Just add a semicolon (;) at the end of the list of values, and then add definitions of the methods and variables in the usual way. For example, we might make an enumerated type to represent the possible values of a playing card. It might be useful to have a method that returns the corresponding value in the game of Blackjack. As another example, suppose that when we print out one of the values, we'd like to see something different from the default string representation (the identifier that names the constant). In that case, we can override the `toString()` method in the class to print out a different string representation. This would give something like:

```
public enum CardValue {
```

```

    ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
    NINE, TEN, JACK, QUEEN, KING;

    /**
     * Return the value of this CardValue in the game of Blackjack.
     * Note that the value returned for an ace is 1.
     */
    public int blackjackValue() {
        if (this == JACK || this == QUEEN || this == KING)
            return 10;
        else
            return 1 + ordinal();
    }

    /**
     * Return a String representation of this CardValue, using numbers
     * for the numerical cards and names for the ace and face cards.
     */
    public String toString() {
        switch (this) {
            // "this" is one of the enumerated type values
            case ACE:
                // ordinal number of ACE
                return "Ace";
            case JACK:
                // ordinal number of JACK
                return "Jack";
            case QUEEN:
                // ordinal number of QUEEN
                return "Queen";
            case KING:
                // ordinal number of KING
                return "King";
            default:
                // it's a numeric card value
                int numericValue = 1 + ordinal();
                return "" + numericValue;
        }
    }
} // end CardValue

```

The methods `blackjackValue()` and `toString()` are instance methods in *CardValue*. Since `CardValue.JACK` is an object belonging to that class, you can call `CardValue.JACK.blackjackValue()`. Suppose that `cardVal` is declared to be a variable of type *CardValue*, so that it can refer to any of the values in the enumerated type. We can call `cardVal.blackjackValue()` to find the Blackjack value of the *CardValue* object to which `cardVal` refers, and `System.out.println(cardVal)` will implicitly call the method `cardVal.toString()` to obtain the print representation of that *CardValue*. (One other thing to keep in mind is that since *CardValue* is a class, the value of `cardVal` can be `null`, which means it does not refer to any object.)

Remember that `ACE`, `TWO`, ..., `KING` are the only possible objects of type *CardValue*, so in an instance method in that class, `this` will refer to one of those values. Recall that the instance method `ordinal()` is defined in any enumerated type and gives the position of the enumerated type value in the list of possible values, with counting starting from zero.

(If you find it annoying to use the class name as part of the name of every enumerated type constant, you can use static import to make the simple names of the constants directly available—but only if you put the enumerated type into a package. For example, if the enumerated type *CardValue* is defined in a package named `cardgames`, then you could place

```
import static cardgames.CardValue.*;
```

at the beginning of a source code file. This would allow you, for example, to use the name `JACK` in that file instead of `CardValue.JACK`.)

Exercises for Chapter 5

1. In all versions of the *PairOfDice* class in Section 5.2, the instance variables `die1` and `die2` are declared to be `public`. They really should be private, so that they are protected from being changed from outside the class. Write another version of the *PairOfDice* class in which the instance variables `die1` and `die2` are `private`. Your class will need “getter” methods that can be used to find out the values of `die1` and `die2`. (The idea is to protect their values from being changed from outside the class, but still to allow the values to be read.) Include other improvements in the class, if you can think of any. Test your class with a short program that counts how many times a pair of dice is rolled, before the total of the two dice is equal to two.
2. A common programming task is computing statistics of a set of numbers. (A statistic is a number that summarizes some property of a set of data.) Common statistics include the mean (also known as the average) and the standard deviation (which tells how spread out the data are from the mean). I have written a little class called *StatCalc* that can be used to compute these statistics, as well as the sum of the items in the dataset and the number of items in the dataset. You can read the source code for this class in the file *StatCalc.java*. If `calc` is a variable of type *StatCalc*, then the following methods are defined:

- `calc.enter(item);` where `item` is a number, adds the item to the dataset.
- `calc.getCount()` is a function that returns the number of items that have been added to the dataset.
- `calc.getSum()` is a function that returns the sum of all the items that have been added to the dataset.
- `calc.getMean()` is a function that returns the average of all the items.
- `calc.getStandardDeviation()` is a function that returns the standard deviation of the items.

Typically, all the data are added one after the other by calling the `enter()` method over and over, as the data become available. After all the data have been entered, any of the other methods can be called to get statistical information about the data. The methods `getMean()` and `getStandardDeviation()` should only be called if the number of items is greater than zero.

Modify the current source code, *StatCalc.java*, to add instance methods `getMax()` and `getMin()`. The `getMax()` method should return the largest of all the items that have been added to the dataset, and `getMin()` should return the smallest. You will need to add two new instance variables to keep track of the largest and smallest items that have been seen so far.

Test your new class by using it in a program to compute statistics for a set of non-zero numbers entered by the user. Start by creating an object of type *StatCalc*:

```
StatCalc calc; // Object to be used to process the data.
calc = new StatCalc();
```

Read numbers from the user and add them to the dataset. Use 0 as a sentinel value (that is, stop reading numbers when the user enters 0). After all the user’s non-zero

numbers have been entered, print out each of the six statistics that are available from `calc`.

3. This problem uses the *PairOfDice* class from Exercise 5.1 and the *StatCalc* class from Exercise 5.2.

The program in Exercise 4.4 performs the experiment of counting how many times a pair of dice is rolled before a given total comes up. It repeats this experiment 10000 times and then reports the average number of rolls. It does this whole process for each possible total (2, 3, ..., 12).

Redo that exercise. But instead of just reporting the average number of rolls, you should also report the standard deviation and the maximum number of rolls. Use a *PairOfDice* object to represent the dice. Use a *StatCalc* object to compute the statistics. (You'll need a new *StatCalc* object for each possible total, 2, 3, ..., 12. You can use a new pair of dice if you want, but it's not necessary.)

4. The *BlackjackHand* class from Subsection 5.5.1 is an extension of the *Hand* class from Section 5.4. The instance methods in the *Hand* class are discussed in that section. In addition to those methods, *BlackjackHand* includes an instance method, `getBlackjackValue()`, that returns the value of the hand for the game of Blackjack. For this exercise, you will also need the *Deck* and *Card* classes from Section 5.4.

A Blackjack hand typically contains from two to six cards. Write a program to test the *BlackjackHand* class. You should create a *BlackjackHand* object and a *Deck* object. Pick a random number between 2 and 6. Deal that many cards from the deck and add them to the hand. Print out all the cards in the hand, and then print out the value computed for the hand by `getBlackjackValue()`. Repeat this as long as the user wants to continue.

In addition to *TextIO.java*, your program will depend on *Card.java*, *Deck.java*, *Hand.java*, and *BlackjackHand.java*.

5. Write a program that lets the user play Blackjack. The game will be a simplified version of Blackjack as it is played in a casino. The computer will act as the dealer. As in the previous exercise, your program will need the classes defined in *Card.java*, *Deck.java*, *Hand.java*, and *BlackjackHand.java*. (This is the longest and most complex program that has come up so far in the exercises.)

You should first write a subroutine in which the user plays one game. The subroutine should return a **boolean** value to indicate whether the user wins the game or not. Return **true** if the user wins, **false** if the dealer wins. The program needs an object of class *Deck* and two objects of type *BlackjackHand*, one for the dealer and one for the user. The general object in Blackjack is to get a hand of cards whose value is as close to 21 as possible, without going over. The game goes like this.

- First, two cards are dealt into each player's hand. If the dealer's hand has a value of 21 at this point, then the dealer wins. Otherwise, if the user has 21, then the user wins. (This is called a "Blackjack".) Note that the dealer wins on a tie, so if both players have Blackjack, then the dealer wins.
- Now, if the game has not ended, the user gets a chance to add some cards to her hand. In this phase, the user sees her own cards and sees **one** of the dealer's two cards. (In a casino, the dealer deals himself one card face up and one card face down. All the user's cards are dealt face up.) The user makes a decision whether to "Hit",

which means to add another card to her hand, or to “Stand”, which means to stop taking cards.

- If the user Hits, there is a possibility that the user will go over 21. In that case, the game is over and the user loses. If not, then the process continues. The user gets to decide again whether to Hit or Stand.
- If the user Stands, the game will end, but first the dealer gets a chance to draw cards. The dealer only follows rules, without any choice. The rule is that as long as the value of the dealer’s hand is less than or equal to 16, the dealer Hits (that is, takes another card). The user should see all the dealer’s cards at this point. Now, the winner can be determined: If the dealer has gone over 21, the user wins. Otherwise, if the dealer’s total is greater than or equal to the user’s total, then the dealer wins. Otherwise, the user wins.

Two notes on programming: At any point in the subroutine, as soon as you know who the winner is, you can say “`return true;`” or “`return false;`” to end the subroutine and return to the main program. To avoid having an overabundance of variables in your subroutine, remember that a function call such as `userHand.getBlackjackValue()` can be used anywhere that a number could be used, including in an output statement or in the condition of an `if` statement.

Write a main program that lets the user play several games of Blackjack. To make things interesting, give the user 100 dollars, and let the user make bets on the game. If the user loses, subtract the bet from the user’s money. If the user wins, add an amount equal to the bet to the user’s money. End the program when the user wants to quit or when she runs out of money.

An applet version of this program can be found in the on-line version of this exercise. You might want to try it out before you work on the program.

6. Subsection 5.7.6 discusses the possibility of representing the suits and values of playing cards as enumerated types. Rewrite the *Card* class from Subsection 5.4.2 to use these enumerated types. Test your class with a program that prints out the 52 possible playing cards. Suggestions: You can modify the source code file *Card.java*, but you should leave out support for Jokers. In your main program, use nested `for` loops to generate cards of all possible suits and values; the `for` loops will be “for-each” loops of the type discussed in Subsection 3.4.4. It would be nice to add a `toString()` method to the *Suit* class from Subsection 5.7.6, so that a suit prints out as “Spades” or “Hearts” instead of “SPADES” or “HEARTS”.

Quiz on Chapter 5

1. Object-oriented programming uses *classes* and *objects*. What are classes and what are objects? What is the relationship between classes and objects?
2. Explain carefully what *null* means in Java, and why this special value is necessary.
3. What is a *constructor*? What is the purpose of a constructor in a class?
4. Suppose that `Kumquat` is the name of a class and that `fruit` is a variable of type `Kumquat`. What is the meaning of the statement “`fruit = new Kumquat();`”? That is, what does the computer do when it executes this statement? (Try to give a complete answer. The computer does several things.)
5. What is meant by the terms *instance variable* and *instance method*?
6. Explain what is meant by the terms *subclass* and *superclass*.
7. Modify the following class so that the two instance variables are **private** and there is a getter method and a setter method for each instance variable:

```
public class Player {  
    String name;  
    int score;  
}
```
8. Explain why the class *Player* that is defined in the previous question has an instance method named `toString()`, even though no definition of this method appears in the definition of the class.
9. Explain the term *polymorphism*.
10. Java uses “garbage collection” for memory management. Explain what is meant here by garbage collection. What is the alternative to garbage collection?
11. For this problem, you should write a very simple but complete class. The class represents a counter that counts 0, 1, 2, 3, 4, The name of the class should be **Counter**. It has one **private** instance variable representing the value of the counter. It has two instance methods: `increment()` adds one to the counter value, and `getValue()` returns the current counter value. Write a complete definition for the class, **Counter**.
12. This problem uses the **Counter** class from the previous question. The following program segment is meant to simulate tossing a coin 100 times. It should use two **Counter** objects, `headCount` and `tailCount`, to count the number of heads and the number of tails. Fill in the blanks so that it will do so:

```
Counter headCount, tailCount;
tailCount = new Counter();
headCount = new Counter();
for ( int flip = 0; flip < 100; flip++ ) {
    if (Math.random() < 0.5)    // There's a 50/50 chance that this is true.
        _____ ;    // Count a "head".
    else
        _____ ;    // Count a "tail".
}

System.out.println("There were " + _____ + " heads.");
System.out.println("There were " + _____ + " tails.");
```