SESSION 4

Programming Languages for Objects

- Black Boxes
- Static Subroutines and Static Variables
- Parameters
- Return Values
- APIs, Packages, and Javadoc
- More on Program Design
- The Truth About Declarations

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 though 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 mobile phone, 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 mobile phone, although the interface in that case is a lot more complicated.

Now, a black box does have an inside -- the code in a subroutine that actually performs the task, or 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 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 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

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," as we will see later) helps control the confusion that might result from so many different names.

There is a basic distinction in Java between static and non-static subroutines. A class definition can contain the source code for both types of subroutine, but what's done with them when the program runs is very different. Static subroutines are easier to understand: In a running program, a static subroutine is a member of the class itself. Non-static subroutine definitions, on the other hand, are only there to be used when objects are created, and the subroutines themselves become members of the objects. Non-static subroutines only become relevant when you are working with objects. The distinction between static and non-static also applies to variables and to other things that can occur in class definitions. This chapter will deal with static subroutines and static variables almost exclusively. We'll turn to non-static stuff and to object-oriented programming in the next chapter.

A subroutine that is in a class or object 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" in this chapter, at least for static subroutines. However, you should start thinking of the terms "method" and "subroutine" as being essentially synonymous as far as Java is concerned.

4.2.1 Subroutine Definitions

A subroutine must be defined somewhere. The definition has to include the name of the subroutine, enough information to make it possible to call the subroutine, and the code that will be executed each time the subroutine is called. 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 `drawFrame()` routine of the animation programs in Section 3.9. 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. It can be a type name such as *String* or int or even an array type such as `double[]`. 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 mentioned in Subsection 2.6.6, 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`". In this case, the type for the parameter is the array type `String[]`.

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 objects 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. The number of parameters that you provide when you call a subroutine must match the number listed in the parameter list in the subroutine definition, and the types of the parameters in the call statement must match the types in the subroutine definition.

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 routine:

```
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;
    System.out.println();
    System.out.print("What is your first guess? ");
    while (true) {
       usersGuess = TextIO.getInt();  // Get the user's guess.
       guessCount++;
       if (usersGuess == computersNumber) {
          System.out.println("You got it in " + guessCount
                  + " guesses!  My number was " + computersNumber);
          break;  // The game is over; the user has won.
       }
       if (guessCount == 6) {
          System.out.println("You didn't get the number in 6
guesses.");
          System.out.println("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)
          System.out.print("That's too low.  Try again: ");
       else if (usersGuess > computersNumber)
          System.out.print("That's too high.  Try again: ");
    }
    System.out.println();
} // 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 its definition, only a call statement. 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) {
      System.out.println("Let's play a game.  I'll pick a number 
between");
      System.out.println("1 and 100, and you try to guess it.");
      boolean playAgain;
      do {
         playGame();  // call subroutine to play one game
         System.out.print("Would you like to play again? ");
         playAgain = TextIO.getlnBoolean();
      } while (playAgain);
      System.out.println("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;
       System.out.println();
       System.out.print("What is your first guess? ");
       while (true) {
          usersGuess = TextIO.getInt();  // Get the user's guess.
          guessCount++;
          if (usersGuess == computersNumber) {
             System.out.println("You got it in " + guessCount
                     + " guesses!  My number was " + 
computersNumber);
             break;  // The game is over; the user has won.
          }
          if (guessCount == 6) {
             System.out.println("You didn't get the number in 6 
guesses.");
             System.out.println("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)
             System.out.print("That's too low.  Try again: ");
          else if (usersGuess > computersNumber)
             System.out.print("That's too high.  Try again: ");
       }
       System.out.println();
   } // 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. Another term for them is global variable.

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 as a whole, 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 usersName;
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`. Similarly, `Math.PI` is a public static member variable in the *Math*. If `numberOfPlayers` is a public static member variable in a class

named Poker, then code in the Poker class would refer to it simply as numberOfPlayers, while code in another class would refer to it as Poker.numberOfPlayers.

As an example, let's add a couple static member variable to the GuessingGame class that we wrote earlier in this section. We add a variable named gamesPlayed to keep track of how many games the user has played and another variable named gamesWon to keep track of the number of games that the user has won. The variables are declared as static member variables:

```
static int gamesPlayed;
static int gamesWon;
```

In the playGame() routine, we always add 1 to gamesPlayed, and we add 1 to gamesWon if the user wins the game. At the end of the main() routine, we print out the values of both variables. It would be impossible to do the same thing with local variables, since both subroutines need to access the variables, and local variables exist in only one subroutine.

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. The default values are the same as those that are used when initializing the elements of an array: For numeric variables, the default value is zero; for boolean variables, the default is false; for char variables, it's the character that has Unicode code number zero; and for objects, such as *Strings*, the default initial value is the special value null.

Since they are of type int, the static member variables gamesPlayed and gamesWon automatically get zero as their initial value. This happens to be the correct initial value for a variable that is being used as a counter. You can, of course, assign a value to a variable at the beginning of the main() routine if you are not satisfied with the default initial value, or if you want to emphasize that you are depending on the default.

Here's the revised version of GuessingGame.java. The changes from the above version are shown in red:

```
public class GuessingGame2 {

    static int gamesPlayed;   // The number of games played.
    static int gamesWon;      // The number of games won.

    public static void main(String[] args) {
       gamesPlayed = 0;
       gamesWon = 0;  // This is actually redundant, since 0 is
                      //                 the default initial value.
       System.out.println("Let's play a game.  I'll pick a number 
between");
       System.out.println("1 and 100, and you try to guess it.");
       boolean playAgain;
       do {
          playGame();  // call subroutine to play one game
          System.out.print("Would you like to play again? ");
          playAgain = TextIO.getlnBoolean();
```

```
       } while (playAgain);
       System.out.println();
       System.out.println("You played " + gamesPlayed + " games,");
       System.out.println("and you won " + gamesWon + " of those
games.");
       System.out.println("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.
        gamesPlayed++;  // Count this game.
        computersNumber = (int)(100 * Math.random()) + 1;
                 // The value assigned to computersNumber is a
randomly
                 //    chosen integer between 1 and 100, inclusive.
        guessCount = 0;
        System.out.println();
        System.out.print("What is your first guess? ");
        while (true) {
           usersGuess = TextIO.getInt();  // Get the user's guess.
           guessCount++;
           if (usersGuess == computersNumber) {
              System.out.println("You got it in " + guessCount
                      + " guesses!  My number was " +
computersNumber);
              gamesWon++;  // Count this win.
              break;       // The game is over; the user has won.
           }
           if (guessCount == 6) {
              System.out.println("You didn't get the number in 6
guesses.");
              System.out.println("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)
              System.out.print("That's too low.  Try again: ");
           else if (usersGuess > computersNumber)
              System.out.print("That's too high.  Try again: ");
        }
        System.out.println();
    } // end of playGame()

} // end of class GuessingGame2
```

(By the way, notice that in my example programs, I didn't mark the static subroutines or variables as being `public` or `private`. You might wonder what it means to leave out both

modifiers. Recall that global variables and subroutines with no access modifier can be used anywhere in the same package as the class where they are defined, but not in other packages. Classes that don't declare a package are in the default package. So, any class in the default package would have access to `gamesPlayed`, `gamesWon`, and `playGame()` -- and that includes pretty much every class in this book. In fact, it is considered to be good practice to make member variables and subroutines `private`, unless there is a reason for doing otherwise.)

Parameters

IF A SUBROUTINE IS A BLACK BOX, then a parameter is something that 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 it by 3 and add 1; if N is even, divide it 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 can 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.

   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. But notice that there is nothing in the subroutine definition that gives a value to the parameter! The parameter gets its initial value from **outside** the subroutine. When the subroutine is called, a value must be provided for the parameter in the subroutine call statement. This value will be assigned to `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 first 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 the subroutine can be called by saying "`print3NSequence(K);`". When the computer executes this subroutine call statement, it takes the value of the variable `K`, assigns that value to `startingValue`, and then executes 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) {
   System.out.println("This program will print out 3N+1
sequences");
   System.out.println("for starting values that you specify.");
   System.out.println();
   int K;  // Input from user; loop ends when K < 0.
   do {
      System.out.println("Enter a starting value.");
      System.out.print("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, *String*, or `double[]`. 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 beginner's 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 have already been assigned initial values! The computer automatically assigns values to the parameters before it starts executing the code inside the subroutine. 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 subroutine call statement'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: `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. 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 with *System.out*. This object includes many different methods named `println`, for example. These methods all have different signatures, such as:

```
println(int)                   println(double)
println(char)                  println(boolean)
println()
```

The computer knows which of these subroutines you want to use based on the type of the actual parameter that you provide. `System.out.println(17)` calls the subroutine with signature `println(int)`, while `System.out.println('A')` calls the subroutine with signature `println(char)`. 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 char, which is different from printing out a boolean, and so forth -- so that each of these operations requires a different subroutine.

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 subroutines defined as:

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

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

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 )  // Dose D evenly divide N?
          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 private 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, and we are told that the subroutine is private, 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.
 */
private 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 should 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.
 */
private 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.
    System.out.print("Enter a line of text: ");
    inputLine = TextIO.getln();
    System.out.println();
    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 Array Parameters

It's possible for the type of a parameter to be an array type. This means that an entire array of values can be passed to the subroutine as a single parameter. For example, we might want a subroutine to print all the values in an integer array in a neat format, separated by commas and

enclosed in a pair of square brackets. To tell it which array to print, the subroutine would have a parameter of type `int[]`:

```
static void printValuesInList( int[] list ) {
    System.out.print('[');
    int i;
    for ( i = 0; i < list.length; i++ ) {
        if ( i > 0 )
            System.out.print(','); // No comma in front of list[0]
        System.out.print(list[i]);
    }
    System.out.println((']');
}
```

To use this subroutine, you need an actual array. Here is a legal, though not very realistic, code segment that creates an array just to pass it as an argument to the subroutine:

```
int[] numbers;
numbers = new int[3];
numbers[0] = 42;
numbers[1] = 17;
numbers[2] = 256;
printValuesInList( numbers );
```

The output produced by the last statement would be `[42,17,256]`.

4.3.6 Command-line Arguments

The `main` routine of a program has a parameter of type `String[]`. When the main routine is called, some actual array of *String* must be passed to `main()` as the value of the parameter. The system provides the actual parameter when it calls `main()`, so the values come from outside the program. Where do the strings in the array come from, and what do they mean? The strings in the array are command-line arguments from the command that was used to run the program. When using a command-line interface, the user types a command to tell the system to execute a program. The user can include extra input in this command, beyond the name of the program. This extra input becomes the command-line arguments. The system takes the command-line arguments, puts them into an array of strings, and passes that array to `main()`.

For example, if the name of the program is `myProg`, then the user can type "`java myProg`" to execute the program. In this case, there are no command-line arguments. But if the user types the command

```
java myProg one two three
```

then the command-line arguments are the strings "one", "two", and "three". The system puts these strings into an array of *Strings* and passes that array as a parameter to the `main()` routine.

Here, for example, is a short program that simply prints out any command line arguments entered by the user:

```
public class CLDemo {

   public static void main(String[] args) {
      System.out.println("You entered " + args.length
                                  + " command-line arguments");
      if (args.length > 0) {
         System.out.println("They were:");
         for (int i = 0; i < args.length; i++)
            System.out.println("   " + args[i]);
      }
   } // end main()

} // end class CLDemo
```

Note that the parameter, `args`, can be an array of length zero. This just means that the user did not include any command-line arguments when running the program.

In practice, command-line arguments are often used to pass the names of files to a program. For example, consider the following program for making a copy of a text file. It does this by copying one line at a time from the original file to the copy, using TextIO. The function `TextIO.eof()` is a boolean-valued function that is `true` if the end of the file has been reached.

```
/**
 *  Requires two command line arguments, which must be file names.
The
 *  the first must be the name of an existing file.  The second is
the name
 *  of a file to be created by the program.  The contents of the
first file
 *  are copied into the second.  WARNING:  If the second file
already
 *  exists when the program is run, its previous contents will be
lost!
 *  This program only works for plain text files.
 */
public class CopyTextFile {

    public static void main( String[] args ) {
        if (args.length < 2 ) {
            System.out.println("Two command-line arguments are
required!");
            System.exit(1);
        }
        TextIO.readFile( args[0] );   // Open the original file for
reading.
        TextIO.writeFile( args[1] );  // Open the copy file for
writing.
        int lineCount;  // Number of lines copied
        lineCount = 0;
        while ( TextIO.eof() == false ) {
```

```
                // Read one line from the original file and write it to
the copy.
                String line;
                line = TextIO.getln();
                TextIO.putln(line);
                lineCount++;
            }
            System.out.printf( "%d lines copied from %s to %s%n",
                                    lineCount, args[0], args[1] );
        }

}
```

Since most programs are run in a GUI environment these days, command-line arguments aren't as important as they used to be. But at least they provide a nice example of how array parameters can be used.

4.3.7 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 the 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 *IllegalArgumentExceptions* in response to bad parameter values. You might want to do the same 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. For this to work, the subroutine call would have to be in the "try" part of the statement.

4.3.8 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.

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 arrays and 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. Such a variable is said to be global to the subroutine, as opposed to the local variables defined inside the subroutine. A global variable can be used in the entire class in which it is defined and, if it not `private`, in other classes as well. 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 values of the global variables, `gamesPlayed` and `gamesWon`, are computed inside a subroutine and are 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.

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 assignment statement, which then has the same effect as the statement "`totalLength = 17+13.0`". The return value is added to `17`, and the result, 30.0, is stored in the variable, `totalLength`.

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.

Note that a `return` inside a loop will end the loop as well as the subroutine that contains it. Similarly, a `return` in a `switch` statement breaks out of the `switch` statement as well as the subroutine. So, you will sometimes use `return` in contexts where you are used to seeing a `break`.

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 the 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;

   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) {
       N = nextN( N );   // Compute next term, using the function
nextN.
       count++;           // Count this term.
       System.out.println(N);  // Print this term.
   }

   System.out.println();
   System.out.println("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:

```
/**
 * This 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. The 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) {

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

      int K;   // Starting point for sequence, specified by the
user.
      do {
         System.out.println("Enter a starting value;");
         System.out.print("To end the program, enter 0: ");
         K = TextIO.getlnInt();  // 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.

      System.out.println("The 3N+1 sequence starting from " + N);
      System.out.println();
      System.out.printf("%8d", N);  // 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
             System.out.println();  // ...then output a carriage
return
             onLine = 0;      // ...and note that there are no
terms
                              //               on the new line.
          }
          System.out.printf("%8d", N);  // Print this term in an 8-
char column.
          onLine++;   // Add 1 to the number of terms on this line.
      }

      System.out.println();  // end current line of output
      System.out.println();  // and then add a blank line
      System.out.println("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.

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 wanted to program for Macintosh computers -- and to produce programs that look and behave the way users expect them to -- had to 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 Application 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 Mac OS, 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

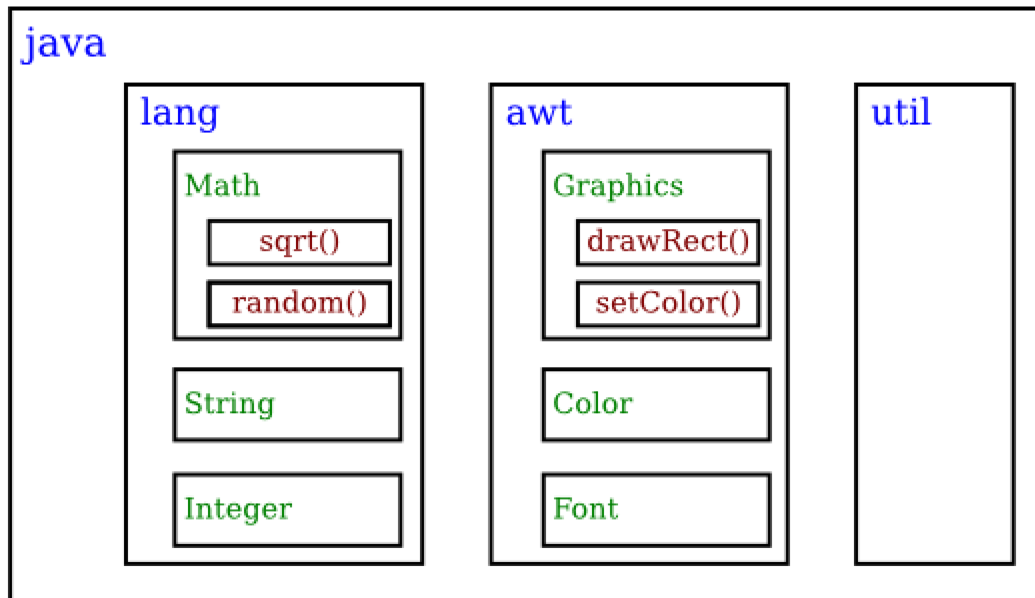
Like all subroutines in Java, the routines in the standard API are grouped into classes. To provide larger-scale organization, classes in Java can be grouped into packages, which were introduced briefly in Subsection 2.6.6. You can have even higher levels of grouping, since packages can also contain 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. AWT is the older of the two Java GUI toolboxes and 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 GUI 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:



Subroutines nested in classes nested in two layers of packages.
The full name of sqrt() is java.lang.Math.sqrt().

The official documentation for the standard Java 7 API lists 209 different packages, including sub-packages, and it lists 4024 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://download.oracle.com/javase/7/docs/api/

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, and you will hardly ever see full names like `java.awt.Color` used in a program. 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, which is `Color`. Note that the `import` line comes at the start of a file (after the `package` statement, if there is one) 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. The `import` directive "`import java.awt.Color`" 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; for example, 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 often 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 a class 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. You can still use both classes in your program: 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, the source code file would be placed in a folder with the same name as the package, "utilities" in this example. And a class that is in a subpackage must be in a subfolder. For example, a class in a package named `utilities.net` would be in folder named "net" inside a folder named "utilities". 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, with just a couple of exceptions, I will not be creating 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 recent standard versions, they are stored in jar files in a subdirectory named `lib` inside the Java Runtime Environment 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 `rt.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. Almost 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 many of the examples in this book.

Note that a Javadoc comment must be 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* by following this link: TextIO Javadoc documentation. 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.

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. The book will not teach you HTML, 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. Furthermore, the characters & and< have special meaning in HTML and should not be used in Javadoc comments except with those meanings; they can be written as `&` and `<`.)

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 four tags: `@author`, `@param`, `@return`, and `@throws`. The `@author` tag can be used only for a class, and should be followed by the name of the author. The other three tags are used in Javadoc comments for a subroutine to provide information about its parameters, its return value, and the exceptions that it might throw. These tags **must** be 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 doc 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 use Javadoc comments for many 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 integrated development environments that were also discussed in Section 2.6. I won't go into any of the details here; consult the documentation for your programming environment.

4.5.5 Static Import

Before ending this section, I will mention an extension of the `import` directive. We have seen that `import` makes it possible to refer to a class such as `java.awt.Color` using its simple name, *Color*. 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`.

There is another 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. That 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 say `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 want to do that, you have to move *TextIO* into a package.

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 program code. 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.

A 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(x)` 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. Or if it only makes sense to call the subroutine at certain times, the precondition might refer to the state that the program must be in when the subroutine is called.

The postcondition of a subroutine, on the other hand, specifies the task that it performs. For a function, the postcondition should specify the value that the function returns.

Subroutines are sometimes 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 pre-written subroutines as building blocks and we will also design new subroutines that we need to complete the project. The API that I will use here is defined in *Mosaic.java*, which in turns depends on *MosaicPanel.java*. To compile and run a program that uses the API, the classes *Mosaic* and *MosaicPanel* must be available. That is, the files `Mosaic.java` and `MosaicPanel.java`, or the the corresponding compiled class files, must be in the same folder as the class that defines the program.

So, 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. (Remeber that a Javadoc comment comes before the thing that it is commenting on.)

```
/**
 * 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()`.

You'll notice that the comments on the subroutine don't specify what happens when the preconditions are **not** met. Although a subroutine is not really obligated by its contract to do anything particular in that case, it would be good to know what happens. For example, if the precondition, "row and col are in the valid range of row and column numbers," on the `setColor()` or `getRed()` routine is violated, an *IllegalArgumentException* will be thrown. Knowing that fact would allow you to write programs that catch and handle the exception, and it would be good to document it with a `@throws` doc tag in the Javadoc comment. Other questions remain about the behavior of the subroutines. For example, what happens if you call `Mosaic.open()` and there is already a mosaic window open on the screen? (In fact, the old one will be closed, and a new one will be created.) It's difficult to fully document the behavior of a piece of software -- sometimes, you just have to experiment or look at the full source code.

My idea for a program 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 new squares and changing the color of those squares. 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 16 rows and 20 columns of squares in my mosaic, so setting the current position to be in the center means setting `currentRow` to 8 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(16,20,25,25)
fillWithRandomColors();
currentRow = 8;       // 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 much, much too fast, the line "`Mosaic.delay(1);`" 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 < 16; 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 (which would violate a precondition of several of the *Mosaic* subroutines). Rather than let this happen, I decide to move the disturbance to the opposite edge of the grid by setting `currentRow` to 15. (Remember that the 16 rows are numbered from 0 to 15.) An alternative to jumping to the opposite edge would be to simply do nothing in this case. 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 = 15;    // 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 >= 16)
         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. You can find a copy of the source code in *RandomMosaicWalk.java*. Remember that this program actually depends on two other files, *Mosaic.java* and *MosaicPanel.java*.

```
/**
 * 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 disturbance in a random walk around the
window
     * as long as the window is open.
     */
    public static void main(String[] args) {
        Mosaic.open(16,20,25,25);
        fillWithRandomColors();
        currentRow = 8;   // start at center of window
        currentColumn = 10;
        while (Mosaic.isOpen()) {
            changeToRandomColor(currentRow, currentColumn);
            randomMove();
            Mosaic.delay(1);
        }
    }  // 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 < 16; 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 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 = 15;
            break;
        case 1:  // move right
            currentColumn++;
            if (currentColumn >= 20)
                currentColumn = 0;
            break;
        case 2:  // move down
            currentRow ++;
            if (currentRow >= 16)
                currentRow = 0;
            break;
        case 3:  // move left
            currentColumn--;
            if (currentColumn < 0)
                currentColumn = 19;
            break;
        }
    }  // end randomMove

} // end class RandomMosaicWalk
```

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 from now on.

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);
}
```

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);
   }
}
```

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 {
   private static double interestRate = 0.05;
   private 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 {
   private 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;`".

Even array variables can be initialized. An array contains several elements, not just a single value. To initialize an array variable, you can provide a list of values, separated by commas, and enclosed between a pair of braces. For example:

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13, 17, 23, 29 };
```

In this statement, an array of int of length 9 is created and filled with the values in the list. The length of the array is determined by the number of items in the list.

Note that this syntax for initializing arrays **cannot** be used in assignment statements. It can only be used in a declaration statement at the time when the array variable is declared.

It is also possible to initialize an array variable with an array created using the `new` operator (which **can** also be used in assignment statements). For example:

```
String[] namelist = new String[100];
```

but in that case, of course, all the array elements will have their default value.

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 0.05 is meant to be the value throughout the entire program. In that 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

```
public 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. (A "final" modifier on a public interest rate makes a lot of sense -- a bank might want to publish its interest rate, but it certainly wouldn't want to let random people make changes to it!)

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

```
public 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 ALIGNMNENT_RIGHT = 1;
public static final int ALIGNMENT_CENTER = 2;
```

In fact, this is how things were generally done before the introduction of enumerated types, 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, say, 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 = 20;        // 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 constant, 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 red. I've left out some of the comments to save space.

```
public class RandomMosaicWalk2 {

    final static int ROWS = 20;        // 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 the
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(1);
         }
    }  // 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 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 in Chapter 9. If the subroutine is not `private`, it can also be accessed from outside the class where it is defined, using its full name.

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, 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 instead of the local 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

className.variableName. (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 the name of the formal parameter. However, please remember that not everything that is possible is a good idea!