Section 4.2

Static Subroutines and Static Variables

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

[Previous Section | Next Section | Chapter Index | Main Index]