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

Section 3.3

The while and do..while Statements

Subsections

STATEMENTS IN JAVA CAN be either simple statements or compound statements. Simple statements, such as assignment statements and subroutine call statements, are the basic building blocks of a program. Compound statements, such as `while` loops and `if` statements, are used to organize simple statements into complex structures, which are called control structures because they control the order in which the statements are executed. The next five sections explore the details of control structures that are available in Java, starting with the `while` statement and the `do..while` statement in this section. At the same time, we'll look at examples of programming with each control structure and apply the techniques for designing algorithms that were introduced in the previous section.

3.3.1 The while Statement

The `while` statement was already introduced in Section 3.1. A `while` loop has the form

```
while ( boolean-expression )
   statement
```

The **statement** can, of course, be a block statement consisting of several statements grouped together between a pair of braces. This statement is called the body of the loop. The body of the loop is repeated as long as the **boolean-expression** is true. This boolean expression is called the continuation condition, or more simply the test, of the loop. There are a few points that might need some clarification. What happens if the condition is false in the first place, before the body of the loop is executed even once? In that case, the body of the loop is never executed at all. The body of a while loop can be executed any number of times, including zero. What happens if the condition is true, but it becomes false somewhere in the **middle** of the loop body? Does the loop end as soon as this happens? It doesn't, because the computer continues executing the body of the loop until it gets to the end. Only then does it jump back to the beginning of the loop and test the condition, and only then can the loop end.

Let's look at a typical problem that can be solved using a `while` loop: finding the average of a set of positive integers entered by the user. The average is the sum of the integers, divided by the number of integers. The program will ask the user to enter one integer at a time. It will keep count of the number of integers entered, and it will keep a running total of all the numbers it has read so far. Here is a pseudocode algorithm for the program:

```
Let sum = 0     // The sum of the integers entered by the user.
Let count = 0   // The number of integers entered by the user.
while there are more integers to process:
    Read an integer
    Add it to the sum
    Count it
Divide sum by count to get the average
Print out the average
```

But how can we test whether there are more integers to process? A typical solution is to tell the user to type in zero after all the data have been entered. This will work because we are assuming that all

the data are positive numbers, so zero is not a legal data value. The zero is not itself part of the data to be averaged. It's just there to mark the end of the real data. A data value used in this way is sometimes called a sentinel value. So now the test in the while loop becomes "while the input integer is not zero". But there is another problem! The first time the test is evaluated, before the body of the loop has ever been executed, no integer has yet been read. There is no "input integer" yet, so testing whether the input integer is zero doesn't make sense. So, we have to do something **before** the while loop to make sure that the test makes sense. Setting things up so that the test in a `while` loop makes sense the first time it is executed is called priming the loop. In this case, we can simply read the first integer before the beginning of the loop. Here is a revised algorithm:

```
Let sum = 0
Let count = 0
Read an integer
while the integer is not zero:
    Add the integer to the sum
    Count it
    Read an integer
Divide sum by count to get the average
Print out the average
```

Notice that I've rearranged the body of the loop. Since an integer is read before the loop, the loop has to begin by processing that integer. At the end of the loop, the computer reads a new integer. The computer then jumps back to the beginning of the loop and tests the integer that it has just read. Note that when the computer finally reads the sentinel value, the loop ends before the sentinel value is processed. It is not added to the sum, and it is not counted. This is the way it's supposed to work. The sentinel is not part of the data. The original algorithm, even if it could have been made to work without priming, was incorrect since it would have summed and counted all the integers, including the sentinel. (Since the sentinel is zero, the sum would still be correct, but the count would be off by one. Such so-called off-by-one errors are very common. Counting turns out to be harder than it looks!)

We can easily turn the algorithm into a complete program. Note that the program cannot use the statement "`average = sum/count;`" to compute the average. Since `sum` and `count` are both variables of type int, the value of `sum/count` is an integer. The average should be a real number. We've seen this problem before: we have to convert one of the int values to a double to force the computer to compute the quotient as a real number. This can be done by type-casting one of the variables to type double. The type cast "(double)sum" converts the value of `sum` to a real number, so in the program the average is computed as "`average = ((double)sum) / count;`". Another solution in this case would have been to declare `sum` to be a variable of type double in the first place.

One other issue is addressed by the program: If the user enters zero as the first input value, there are no data to process. We can test for this case by checking whether `count` is still equal to zero after the `while` loop. This might seem like a minor point, but a careful programmer should cover all the bases.

Here is the full source code for the program:

```
/**
 * This program reads a sequence of positive integers input
 * by the user, and it will print out the average of those
 * integers.  The user is prompted to enter one integer at a
 * time.  The user must enter a 0 to mark the end of the
 * data.  (The zero is not counted as part of the data to
 * be averaged.)  The program does not check whether the
 * user's input is positive, so it will actually add up
 * both positive and negative input values.
 */

public class ComputeAverage {
```

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

      int inputNumber;   // One of the integers input by the user.
      int sum;           // The sum of the positive integers.
      int count;         // The number of positive integers.
      double average;    // The average of the positive integers.

      /* Initialize the summation and counting variables. */

      sum = 0;
      count = 0;

      /* Read and process the user's input. */

      System.out.print("Enter your first positive integer: ");
      inputNumber = TextIO.getlnInt();

      while (inputNumber != 0) {
         sum += inputNumber;   // Add inputNumber to running sum.
         count++;              // Count the input by adding 1 to count.
         System.out.print("Enter your next positive integer, or 0 to end: ");
         inputNumber = TextIO.getlnInt();
      }

      /* Display the result. */

      if (count == 0) {
         System.out.println("You didn't enter any data!");
      }
      else {
         average = ((double)sum) / count;
         System.out.println();
         System.out.println("You entered " + count + " positive integers.");
         System.out.printf("Their average is %1.3f.\n", average);
      }

   } // end main()

} // end class ComputeAverage
```

3.3.2 The do..while Statement

Sometimes it is more convenient to test the continuation condition at the end of a loop, instead of at the beginning, as is done in the `while` loop. The `do..while` statement is very similar to the `while` statement, except that the word "while," along with the condition that it tests, has been moved to the end. The word "do" is added to mark the beginning of the loop. A `do..while` statement has the form

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

or, since, as usual, the **statement** can be a block,

```
do {
    statements
} while ( boolean-expression );
```

Note the semicolon, ';', at the very end. This semicolon is part of the statement, just as the semicolon at the end of an assignment statement or declaration is part of the statement. Omitting it is a syntax error. (More generally, **every** statement in Java ends either with a semicolon or a right brace, '}'.)

To execute a do loop, the computer first executes the body of the loop -- that is, the statement or statements inside the loop -- and then it evaluates the boolean expression. If the value of the expression is true, the computer returns to the beginning of the do loop and repeats the process; if the value is false, it ends the loop and continues with the next part of the program. Since the condition is not tested until the end of the loop, the body of a do loop is always executed at least once.

For example, consider the following pseudocode for a game-playing program. The do loop makes sense here instead of a while loop because with the do loop, you know there will be at least one game. Also, the test that is used at the end of the loop wouldn't even make sense at the beginning:

```
do {
   Play a Game
   Ask user if he wants to play another game
   Read the user's response
} while ( the user's response is yes );
```

Let's convert this into proper Java code. Since I don't want to talk about game playing at the moment, let's say that we have a class named Checkers, and that the Checkers class contains a static member subroutine named playGame() that plays one game of checkers against the user. Then, the pseudocode "Play a game" can be expressed as the subroutine call statement "Checkers.playGame();". We need a variable to store the user's response. The TextIO class makes it convenient to use a boolean variable to store the answer to a yes/no question. The input function TextIO.getlnBoolean() allows the user to enter the value as "yes" or "no" (among other acceptable responses). "Yes" is considered to be true, and "no" is considered to be false. So, the algorithm can be coded as

```
boolean wantsToContinue;  // True if user wants to play again.
do {
   Checkers.playGame();
   System.out.print("Do you want to play again? ");
   wantsToContinue = TextIO.getlnBoolean();
} while (wantsToContinue == true);
```

When the value of the boolean variable is set to false, it is a signal that the loop should end. When a boolean variable is used in this way -- as a signal that is set in one part of the program and tested in another part -- it is sometimes called a flag or flag variable (in the sense of a signal flag).

By the way, a more-than-usually-pedantic programmer would sneer at the test "while (wantsToContinue == true)". This test is exactly equivalent to "while (wantsToContinue)". Testing whether "wantsToContinue == true" is true amounts to the same thing as testing whether "wantsToContinue" is true. A little less offensive is an expression of the form "flag == false", where flag is a boolean variable. The value of "flag == false" is exactly the same as the value of "!flag", where ! is the boolean negation operator. So you can write "while (!flag)" instead of "while (flag == false)", and you can write "if (!flag)" instead of "if (flag == false)".

Although a do..while statement is sometimes more convenient than a while statement, having two kinds of loops does not make the language more powerful. Any problem that can be solved using do..while loops can also be solved using only while statements, and vice versa. In fact, if **doSomething** represents any block of program code, then

```
do {
    doSomething
} while ( boolean-expression );
```

has exactly the same effect as

```
doSomething
while ( boolean-expression ) {
```

```
    doSomething
}
```

Similarly,

```
while ( boolean-expression ) {
    doSomething
}
```

can be replaced by

```
if ( boolean-expression ) {
   do {
       doSomething
   } while ( boolean-expression );
}
```

without changing the meaning of the program in any way.

3.3.3 break and continue

The syntax of the `while` and `do..while` loops allows you to test the continuation condition at either the beginning of a loop or at the end. Sometimes, it is more natural to have the test in the middle of the loop, or to have several tests at different places in the same loop. Java provides a general method for breaking out of the middle of any loop. It's called the `break` statement, which takes the form

```
break;
```

When the computer executes a `break` statement in a loop, it will immediately jump out of the loop. It then continues on to whatever follows the loop in the program. Consider for example:

```
while (true) {  // looks like it will run forever!
   System.out.print("Enter a positive number: ");
   N = TextIO.getlnInt();
   if (N > 0)   // the input value is OK, so jump out of loop
      break;
   System.out.println("Your answer must be > 0.");
}
// continue here after break
```

If the number entered by the user is greater than zero, the `break` statement will be executed and the computer will jump out of the loop. Otherwise, the computer will print out "Your answer must be > 0." and will jump back to the start of the loop to read another input value.

The first line of this loop, "`while (true)`" might look a bit strange, but it's perfectly legitimate. The condition in a `while` loop can be any boolean-valued expression. The computer evaluates this expression and checks whether the value is `true` or `false`. The boolean literal "`true`" is just a boolean expression that always evaluates to true. So "`while (true)`" can be used to write an infinite loop, or one that will be terminated by a `break` statement.

A `break` statement terminates the loop that immediately encloses the `break` statement. It is possible to have nested loops, where one loop statement is contained inside another. If you use a `break` statement inside a nested loop, it will only break out of that loop, not out of the loop that contains the nested loop. There is something called a labeled break statement that allows you to specify which loop you want to break. This is not very common, so I will go over it quickly. Labels work like this: You can put a label in front of any loop. A label consists of a simple identifier followed by a colon. For example, a `while` with a label might look like "`mainloop: while...`". Inside this loop you can use the

labeled break statement "`break mainloop;`" to break out of the labeled loop. For example, here is a code segment that checks whether two strings, `s1` and `s2`, have a character in common. If a common character is found, the value of the flag variable `nothingInCommon` is set to `false`, and a labeled break is used to end the processing at that point:

```
boolean nothingInCommon;
nothingInCommon = true;  // Assume s1 and s2 have no chars in common.
int i,j;  // Variables for iterating through the chars in s1 and s2.

i = 0;
bigloop: while (i < s1.length()) {
   j = 0;
   while (j < s2.length()) {
      if (s1.charAt(i) == s2.charAt(j)) { // s1 and s2 have a common char.
          nothingInCommon = false;
          break bigloop;  // break out of BOTH loops
      }
      j++;  // Go on to the next char in s2.
   }
   i++;  //Go on to the next char in s1.
}
```

The `continue` statement is related to `break`, but less commonly used. A `continue` statement tells the computer to skip the rest of the current iteration of the loop. However, instead of jumping out of the loop altogether, it jumps back to the beginning of the loop and continues with the next iteration (including evaluating the loop's continuation condition to see whether any further iterations are required). As with `break`, when a `continue` is in a nested loop, it will continue the loop that directly contains it; a "labeled continue" can be used to continue the containing loop instead.

`break` and `continue` can be used in `while` loops and `do..while` loops. They can also be used in `for` loops, which are covered in the next section. In Section 3.6, we'll see that `break` can also be used to break out of a `switch` statement. A `break` can occur inside an `if` statement, but only if the `if` statement is nested inside a loop or inside a `switch` statement. In that case, it does **not** mean to break out of the `if`. Instead, it breaks out of the loop or `switch` statement that contains the `if` statement. The same consideration applies to `continue` statements inside `if`s.

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