Blocks, Loops, and Branches

THE ABILITY OF A COMPUTER TO PERFORM complex tasks is built on just a few ways of combining simple commands into control structures. In Java, there are six such structures that are used to determine the normal flow of control in a program—and, in fact, just three of them would be enough to write programs to perform any task. The six control structures are: the block, the while loop, the do..while loop, the for loop, the if statement, and the switch statement. Each of these structures is considered to be a single "statement," but a **structured** statement that can contain one or more other statements inside itself.

Blocks

The block is the simplest type of structured statement. Its purpose is simply to group a sequence of statements into a single statement. The format of a block is:

```
{
    statements
}
```

That is, it consists of a sequence of statements enclosed between a pair of braces, "{" and "}". In fact, it is possible for a block to contain no statements at all; such a block is called an empty block, and can actually be useful at times. An empty block consists of nothing but an empty pair of braces. Block statements usually occur inside other statements, where their purpose is to group together several statements into a unit. However, a block can be legally used wherever a statement can occur. There is one place where a block is required: As you might have already noticed in the case of the `main` subroutine of a program, the definition of a subroutine is a block, since it is a sequence of statements enclosed inside a pair of braces.

I should probably note again at this point that Java is what is called a free-format language. There are no syntax rules about how the language has to be arranged on a page. So, for example, you could write an entire block on one line if you want. But as a matter of good programming style, you should lay out your program on the page in a way that will make its structure as clear as possible. In general, this means putting one statement per line and using indentation to indicate statements that are contained inside control structures. This is the format that I will use in my examples.

Here are two examples of blocks:

```
{
   System.out.print("The answer is ");
   System.out.println(ans);
}
```

```
{  // This block exchanges the values of x and y
   int temp;      // A temporary variable for use in this block.
   temp = x;      // Save a copy of the value of x in temp.
   x = y;         // Copy the value of y into x.
   y = temp;      // Copy the value of temp into y.
}
```

In the second example, a variable, `temp`, is declared inside the block. This is perfectly legal, and it is good style to declare a variable inside a block if that variable is used nowhere else but inside the block. A variable declared inside a block is completely inaccessible and invisible from outside that block. When the computer executes the variable declaration statement, it allocates memory to hold the value of the variable (at least conceptually). When the block ends, that memory is discarded (that is, made available for reuse). The variable is said to be local to the block. There is a general concept called the "scope" of an identifier. The scope of an identifier is the part of the program in which that identifier is valid. The scope of a variable defined inside a block is limited to that block, and more specifically to the part of the block that comes after the declaration of the variable.

The Basic While Loop

The block statement by itself really doesn't affect the flow of control in a program. The five remaining control structures do. They can be divided into two classes: loop statements and branching statements. You really just need one control structure from each category in order to have a completely general-purpose programming language. More than that is just convenience.

A while loop is used to repeat a given statement over and over. Of course, it's not likely that you would want to keep repeating it forever. That would be an infinite loop, which is generally a bad thing. To be more specific, a `while` loop will repeat a statement over and over, but only so long as a specified condition remains true.
A `while` loop has the form:

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

Since the statement can be, and usually is, a block, most `while` loops have the form:

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

Some programmers think that the braces should always be included as a matter of style, even when there is only one statement between them, but I don't always follow that advice myself.

The semantics of the `while` statement go like this: When the computer comes to a `while` statement, it evaluates the **boolean-expression**, which yields either `true` or `false` as its value. If the value is `false`, the computer skips over the rest of the `while` loop and proceeds to the next command in the program. If the value of the expression is `true`, the computer executes the **statement** or block of **statements** inside the loop. Then it returns to the beginning of the `while` loop and repeats the process. That is, it re-evaluates the **boolean-expression**, ends the loop if the value is `false`, and continues it if the value is `true`. This will continue over and over until the value of the expression is `false` when the computer evaluates it; if that never happens, then there will be an infinite loop.

Here is an example of a `while` loop that simply prints out the numbers 1, 2, 3, 4, 5:

```
int number;   // The number to be printed.
number = 1;   // Start with 1.
while ( number < 6 ) {  // Keep going as long as number is < 6.
    System.out.println(number);
    number = number + 1;  // Go on to the next number.
}
System.out.println("Done!");
```

The variable `number` is initialized with the value 1. So when the computer evaluates the expression "`number < 6`" for the first time, it is asking whether 1 is less than 6, which is `true`. The computer therefore proceeds to execute the two statements inside the loop. The first statement prints out "1". The second statement adds 1 to `number` and stores the result back into the variable `number`; the value of `number` has been changed to 2. The computer has reached the end of the loop, so it returns to the beginning and asks again whether `number` is less than 6. Once again this is true, so the computer executes the loop again, this time printing out 2 as the value of `number` and then changing the value of `number` to 3. It continues in this way until eventually `number` becomes equal to 6. At that point, the expression "`number < 6`" evaluates to `false`. So, the computer jumps past the end of the

loop to the next statement and prints out the message "Done!". Note that when the loop ends, the value of `number` is 6, but the last value that was printed was 5.

By the way, you should remember that you'll never see a `while` loop standing by itself in a real program. It will always be inside a subroutine which is itself defined inside some class. As an example of a `while` loop used inside a complete program, here is a little program that computes the interest on an investment over several years. This is an improvement over examples from the previous chapter that just reported the results for one year:

```
import textio.TextIO;

/**
 *  This class implements a simple program that will compute the amount of
 *  interest that is earned on an investment over a period of 5 years.
 *  The initial amount of the investment and the interest rate are input
 *  by the user.  The value of the investment at the end of each year is
output.
 */
public class Interest3 {


   public static void main(String[] args) {

      double principal;  // The value of the investment.
      double rate;       // The annual interest rate.

      /* Get the initial investment and interest rate from the user. */

      System.out.print("Enter the initial investment: ");
      principal = TextIO.getlnDouble();

      System.out.println();
      System.out.println("Enter the annual interest rate.");
      System.out.print("Enter a decimal, not a percentage: ");
      rate = TextIO.getlnDouble();
      System.out.println();

      /* Simulate the investment for 5 years. */

      int years;  // Counts the number of years that have passed.

      years = 0;
      while (years < 5) {
         double interest;  // Interest for this year.
         interest = principal * rate;
         principal = principal + interest;     // Add it to principal.
         years = years + 1;    // Count the current year.
         System.out.print("The value of the investment after ");
         System.out.print(years);
         System.out.print(" years is $");
         System.out.printf("%1.2f", principal);
         System.out.println();
      } // end of while loop

   } // end of main()

} // end of class Interest3
```

You should study this program, and make sure that you understand what the computer does step-by-step as it executes the `while` loop.

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.

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 );
```

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.

---

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.

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 and in 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.