Chapter 17

Iteration

Iteration repeats the execution of a sequence of code. Iteration is useful for solving many programming problems. Interation and conditional execution form the basis for algorithm construction.

17.1 The `while` Statement

`CountToFive` (17.1) counts to five by printing a number on each output line.

```
public class CountToFive {
    public void run() {
        System.out.println(1);
        System.out.println(2);
        System.out.println(3);
        System.out.println(4);
        System.out.println(5);
    }
}
```

Listing 17.1: `CountToFive`—Simple count to five

In an Interactions pane we see the results:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> new CountToFive().run();
1
2
3
4
5
>
```

 © 2008 Richard L. Halterman

How would you write the code for CountToTenThousand? Would you copy, paste, and modify 10,000 System.out.println() statements? That would be impractical! Counting is such a common activity, and computers routinely count up to very large values, so there must be a better way. What we really would like to do is print the value of a variable (call it count), then increment the variable (count++), and repeat this process until the variable is large enough (count == 5). This concept is known as *iteration*, and Java has a simple statement that supports it.

IterativeCountToFive (17.2) uses a while statement to count to five:

```
public class IterativeCountToFive {
    public void run() {
        int count = 1;                    //  Initialize counter
        while (count <= 5) {
            System.out.println(count);    // Display counter, then
            count++;                      // Increment counter
        }
    }
}
```

Listing 17.2: IterativeCountToFive—Better count to five

IterativeCountToFive uses a while loop inside its run() method to display a variable that is counting up to five. Unlike with the approach taken in CountToFive, it is trivial to modify IterativeCountToFive to count up to 10,000 (just change the literal value 5 to 10000).

The statments

```
System.out.println(count);  // Display counter, then
count++;                    // Increment counter
```

in IterativeCountToFive constitute the body of the while statement. The statements in the body are repeated over and over until the Boolean condition

```
count <= 5
```

becomes false.

The while statement has the general form:

while (*condition*)
body

- The reserved word while identifies an while statement.
- *condition* is a Boolean expression that determines whether or not the body will be (or will continue to be) executed. The condition must be enclosed within parentheses.
- *body* is like the if statement body (Section 5.2)—it can consist of a single statement or a block of statements.

 © 2008 Richard L. Halterman

Except for the reserved word `while` instead of `if`, `while` statements look identical to `if` statements. Often beginning programmer confuse the two or accidentally type `if` when they mean `while` or vice-versa. Usually the very different behavior of the two statements reveals the problem immediately; however, sometimes, in deeply nested complex logic, this mistake can be hard to detect.

Figure 17.1 shows how program execution flows through a `while` statement.

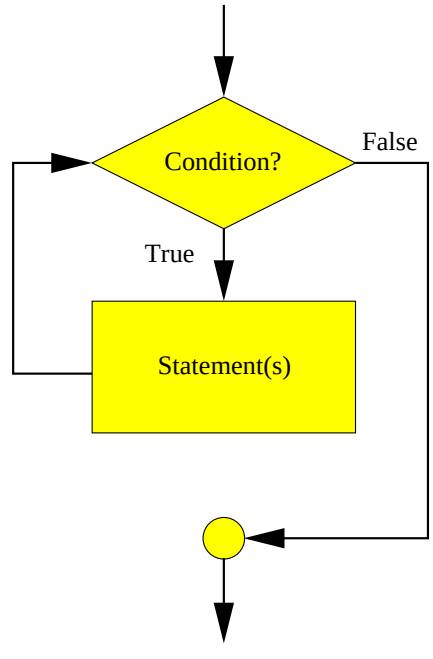

Figure 17.1: Execution flow in a `while` statement

The condition is checked before the body is executed, and then each time after the body has been executed. If the condition is false or becomes false, the body is not executed and the loop terminates. Observe that the body may never be executed if the Boolean expression in the condition is initially false.

`AddUpNonnegatives` (17.3) is a program that allows a user to enter any number of nonnegative integers. When the user enters a negative value, the program no longer accepts input, and it displays the sum of all the nonnegative values. If a negative number is the first entry, the sum is zero.

```
import java.util.Scanner;

public class AddUpNonnegatives {
    public static void main(String[] args) {
        int input = 0,
            sum = 0;
        Scanner scan = new Scanner(System.in);
        while (input >= 0) {
            input = scan.nextInt();
            if (input >= 0) {
                sum += input;
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

 © 2008 Richard L. Halterman

Listing 17.3: AddUpNonnegatives—Sums any number of nonnegative integers

The initialization of input to zero guarantees that the body of the while loop will be executed at least once. The if statement ensures that a negative entry will not be added to sum. (Could the condition have used > instead of >= and achieved the same results?) Upon entry of a negative value, sum will not be updated and the condition will no longer be true. The loop terminates and the print statement is finally executed.

AddUpNonnegatives (17.3) shows that a while loop can be used for more than simple counting. AddUpNonnegatives does not keep track of the number (count) of values entered. The program simply accumulates the entered values in the variable named sum.

It is a little awkward that the same condition appears twice, once in the while and again in the if. The code AddUpNonnegatives (17.3) can be simplified somewhat with some added insight into the assignment statement. A simple assignment statement such as

```
x = y;
```

is actually an expression that has a value. Its value is the same as the value assigned to the variable. That means the following statement is possible:

```
x = (y = z);
```

Here, the value of z is assigned to y via the assignment y = z. This assignment expression itself has the value of z, so z's value is assigned also to x. Unlike the arithmetic binary operators (+, -, *, etc.) which apply from left to right, the assignment operator associates from right to left. This means the parentheses can be omitted:

```
x = y = z;
```

The effect of this statement is that all the variables end up with the same value, z's value. This extended assignment statement is sometimes called a *chained assignment*.

This curious fact about assignment can be put to good use to shorten our AddUpNonnegatives code:

```
import java.util.Scanner;

public class AddUpNonnegativesSimpler {
    public static void main(String[] args) {
        int input, sum = 0;
        Scanner scan = new Scanner(System.in);
        while ((input = scan.nextInt()) >= 0) {
            sum += input;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Listing 17.4: AddUpNonnegativesSimpler—Sums any number of nonnegative integers

 © 2008 Richard L. Halterman

In `AddUpNonnegativesSimpler` (17.4):

- The variable `input` no longer must be initialized to zero, because it is assigned before its value is checked by the `while`. The value of `sum` must still be initialized since it is accumulating the sum and must begin with a zero value.
- In the expression

  ```
  (input = scan.nextInt()) >= 0
  ```

  the assignment is performed first, then the value of the assignment is compared to zero.
- The modification of `sum` can only be performed as long as the condition remains true. The update statement never can be executed whenever `input` is negative; thus, the separate `if` statement is no longer needed.

Some programmers consider the simplified version tricky and prefer the first version. The second version is slightly more efficient since the condition is checked only one time through the loop instead of two. In this case, however, the speed difference is negligible, so the less tricky design is acceptable.

17.2 Nested Loops

Just like in `if` statements, `while` bodies can contain arbitrary Java statements, including other `while` statements. A loop can therefore be nested within another loop. `TimesTable` (17.5) prints a multiplication table on the screen using nested `while` loops.

```
public class TimesTable {
  public static void main(String[] args) {
    //  Print a multiplication table to 10 x 10
    int row = 1;
    //  Print column heading
    System.out.println("       1   2   3   4   5   6   7   8   9  10");
    System.out.println("   +----------------------------------------");
    while ( row <= 10 ) {              //  Table has ten rows.
      System.out.printf("%3d|", row); //  Print heading for this row.
      int column = 1;                  //  Reset column for each row.
      while ( column <= 10 ) {         //  Table has ten columns.
        System.out.printf("%4d", row*column);
        column++;
      }
      row++;
      System.out.println();  //  Move cursor to next row
    }
  }
}
```

Listing 17.5: `TimesTable`—Prints a multiplication table

 © 2008 Richard L. Halterman

The output of `TimesTable` is

```
      1   2   3   4   5   6   7   8   9  10
   +----------------------------------------
  1|   1   2   3   4   5   6   7   8   9  10
  2|   2   4   6   8  10  12  14  16  18  20
  3|   3   6   9  12  15  18  21  24  27  30
  4|   4   8  12  16  20  24  28  32  36  40
  5|   5  10  15  20  25  30  35  40  45  50
  6|   6  12  18  24  30  36  42  48  54  60
  7|   7  14  21  28  35  42  49  56  63  70
  8|   8  16  24  32  40  48  56  64  72  80
  9|   9  18  27  36  45  54  63  72  81  90
 10|  10  20  30  40  50  60  70  80  90 100
```

This is how `TimesTable` works:

- It is important to distinguish what is done only once (outside all loops) from that which is done repeatedly. The column heading across the top of the table is outside of all the loops; therefore, it is printed all at once.
- The work to print the heading for the rows is distributed throughout the execution of the outer loop. This is because the heading for a given row cannot be printed until all the results for the previous row have been printed.
- `System.out.printf()` is used to print the table contents so that the entries are right justified. The field width of four is used because the largest product requires three digits; this ensures that no entries will "run together" and look like one value.
- `row` is the control variable for the outer loop; `column` controls the inner loop.
- The inner loop executes ten times on every single iteration of the outer loop. How many times is the statement

  ```
  System.out.printf("%4d", row*column);
  ```

  executed? $10 \times 10 = 100$, one time for every product in the table.
- A newline is printed (by the only `println()` call inside the loops) after the contents of each row is displayed; thus, all the values printed in the inner (`column`) loop appear on the same line.

17.3 Infinite Loops

An infinite loop is a loop that is never exited. Once the program flow enters the loop's body it cannot escape. Infinite loops are sometimes designed. For example, a long-running server application like a Web server may need to continuously check for incoming queries. This checking can be performed within a loop that runs indefinitely. All

 © 2008 Richard L. Halterman

too often for beginning programmers, however, infinite loops are created by accident and represent a logical error in the program.

Intentional infinite loops should be made obvious. For example,

```
while ( true ) {
    /*  Do something forever . . . */
}
```

The Boolean literal `true` is always true, so it is impossible for the loop's condition to be false. The only ways to exit the loop is via a `break` statement or a `return` (§ 4.1) statement embedded somewhere within its body.

Intentional infinite loops are easy to write correctly. Accidental infinite loops are quite common, but can be puzzling for beginning programmers to diagnose and repair. Consider `FindFactors` (17.6) that proportedly prints all the integers with their associated factors from 1 to 20.

```
public class FindFactors {
    public static void main(String[] args) {
        //  List of the factors of the numbers up to 20
        int n = 1;
        final int MAX = 20;
        while ( n <= MAX ) {
            int factor = 1;
            System.out.print(n + ": ");
            while ( factor <= n ) {
                if ( n % factor == 0 ) {
                    System.out.print(factor + " ");
                    factor++;
                }
            }
            System.out.println();
            n++;
        }
    }
}
```

Listing 17.6: `FindFactors`—an erroneous factoring program

It displays

```
1: 1
2: 1 2
3: 1
```

and then "freezes up" or "hangs," ignoring any user input (except the key sequence Ctrl C on most systems which interrupts and terminates the running JVM program). This type of behavior is a frequent symptom of an unintentional infinite loop. The factors of 1 display properly, as do the factors of 2. The first factor of 3 is properly displayed and

 © 2008 Richard L. Halterman

then the program hangs. Since the program is short, the problem may be easy to locate. In general, though, the logic can be quite complex. Even in `FindFactors` the debugging task is nontrivial since nested loops are involved. (Can you find and fix the problem in `FindFactors` before reading further?)

In order to avoid infinite loops, we must ensure that the loop exhibits certain properties:

- The loop's condition must not be a tautology (a Boolean expression that can never be false). For example,

  ```
  while ( i >= 1 || i <= 10 ) {
      /*  Body omitted  */
  }
  ```

  is an infinite loop since any value chosen for `i` will satisfy one or both of the two subconditions. Most likely the `&&` operator was intended here so the loop continues until `i` is outside the range 1–10.

  In `FindFactors`, the outer loop condition is

  ```
  n <= MAX
  ```

  If `n` is 1 and `MAX` is two, then the condition is false, so this is not a tautology. Checking the inner loop condition:

  ```
  factor <= n
  ```

  we see that if `factor` is 3 and `n` is 2, then the expression is false; therefore, it also is not a tautology.

- The condition of a `while` must be true initially to gain access to its body. The code within the body must modify the state of the program in some way so as to influence the outcome of the condition that is checked at each iteration. This usually means one of the variables used in the condition is modified in the body. Eventually the variable assumes a value that makes the condition false, and the loop terminates.

  In `FindFactors`, the outer loop's condition involves the variable `n` and constant `MAX`. `MAX` cannot change, so it is essential that `n` be modified within the loop. Fortunately, the last statement in the body of the outer loop increments `n`. `n` is initially 1 and `MAX` is 20, so unless the circumstances arise to make the inner loop infinite, the outer loop should eventually terminate.

  The inner loop's condition involves the variables `n` and `factor`. No statement in the inner loop modifies `n`, so it is imperative that `factor` be modified in the loop. The good news is `factor` is incremented in the body of the inner loop, but the bad news is the increment operation is protected within the body of the `if` statement. The inner loop contains one statement, the `if` statement (the `if` statement in turn has two statements in its body):

  ```
  while ( factor <= n ) {
      if ( n % factor == 0 ) {
          System.out.print(factor + " ");
          factor++;
      }
  }
  ```

  If the condition of the `if` is ever false, then the state of the program will not change when the body of the inner loop is executed. This effectively creates an infinite loop. The statement that modifies `factor` must be moved outside of the `if` statement's body:

 © 2008 Richard L. Halterman

```
while ( factor <= n ) {
    if ( n % factor == 0 ) {
        System.out.print(factor + " ");
    }
    factor++;
}
```

This new version runs correctly.

A debugger can be used to step through a program to see where and why an infinite loop arises. Another common technique is to put print statements in strategic places to examine the values of the variables involved in the loop's control. The original inner loop can be so augmented:

```
while ( factor <= n ) {
    System.out.println("factor = " + factor + "  n = " + n);
    if ( n % factor == 0 ) {
        System.out.print(factor + " ");
        factor++;
    }
}
```

It produces the following output:

```
1: factor = 1  n = 1
1
2: factor = 1  n = 2
1 factor = 2  n = 2
2
3: factor = 1  n = 3
1 factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
    .
    .
    .
```

The output demonstrates that once `factor` becomes equal to 2 and `n` becomes equal to 3 the program's execution becomes trapped in the inner loop. Under these conditions:

1. 2 < 3 is true, so the loop continues and
2. 3 `%` 2 is equal to 1, so the `if` will not increment `factor`.

It is imperative that `factor` be incremented each time through the inner loop; therefore, the statement incrementing `factor` must be moved outside of the `if`'s guarded body.

 © 2008 Richard L. Halterman

17.4 Summary

- The `while` allows the execution of code sections to be repeated multiple times.
- The condition of the `while` controls the execution of statements within the `while`'s body.
- The statements within the body of a `while` are executed over and over until the condition of the `while` is false.
- In an infinite loop, the `while`'s condition never becomes false.
- The statements within the `while`'s body must eventually lead to the condition being false; otherwise, the loop will be infinite.
- Infinite loops are rarely intentional and usually accidental.
- An infinite loop can be diagnosed by putting a printing statement inside its body.
- An assignment expression has a value; the expression's value is the same as the expression of the right of the assignment operator.
- Chained assignment allows multiple variables to be assigned the same value in one statement.
- A loop contained within another loop is called a nested loop.
- A loop contained within another loop is called a

17.5 Exercises

1. In `AddUpNonnegatives` (17.3) could the condition of the `if` statement have used `>` instead of `>=` and achieved the same results? Why?
2. In `AddUpNonnegatives` (17.3) could the condition of the `while` statement have used `>` instead of `>=` and achieved the same results? Why?
3. In `AddUpNonnegatives` (17.3) what would happen if the statement assigning `scan` is moved into the loop? Is moving the assignment into the loop a good or bad thing to do? Why?
4. In the following code fragment:

```
int a = 0;
while (a < 100) {
    int b = 0;
    while (b < 55) {
        System.out.println("*");
    }
}
```

   how many asterisks are printed?

5. Modify `TimesTable` (17.5) so that instead of using a `main()` method it uses a `show()` method to print the table. The `show()` method should accept an integer parameter representing the size of the table: 10 prints a 10×10 table, 15 prints a 15×15 table, etc. Accept any integer values from 1 to 18. Be sure everything lines up correctly, and the table looks attractive.

 © 2008 Richard L. Halterman