5.3 `for` Iteration Statement

Section 5.2 presented the essentials of counter-controlled iteration. The `while` statement can be used to implement any counter-controlled loop. Java also provides the `for` **iteration statement**, which specifies the counter-controlled-iteration details in a single line of code. Figure 5.2 reimplements the application of Fig. 5.1 using `for`.

```
// Fig. 5.2: ForCounter.java
// Counter-controlled iteration with the for iter

public class ForCounter {
   public static void main(String[] args) {
      // for statement header includes initializat
      // loop-continuation condition and increment
      for (int counter = 1; counter <= 10; counter
         System.out.printf("%d ", counter);
      }

      System.out.println();
   }
}
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.2

Counter-controlled iteration with the `for` iteration statement.

When the `for` statement (lines 8–10) begins executing, the control variable `counter` is *declared* and *initialized* to `1`. (Recall from Section 5.2 that the first two elements of counter-controlled iteration are the *control variable* and its *initial value*.) Next, the program checks the *loop-continuation condition*, `counter <= 10`, which is between the two required semicolons. Because the initial value of `counter` is `1`, the condition initially is `true`. Therefore, the body statement (line 9) displays control variable `counter`'s value, namely `1`. After executing the loop's body, the program increments `counter` in the expression `counter++`, which appears to the right of the second semicolon. Then the loop-continuation test is performed again to determine whether the program should continue with the next iteration of the loop. At this point, the control variable's value is `2`, so the condition is still `true` (the *final value* is not exceeded)—thus, the program performs the body statement again (i.e., the next iteration of the loop). This process continues until the numbers 1 through 10 have been displayed and the `counter`'s value becomes `11`, causing the loop-continuation test to fail and iteration to terminate (after 10 iterations of the loop body). Then the program performs the first statement after the `for`—in this case, line 12.

Figure 5.2 uses (in line 8) the loop-continuation condition `counter <= 10`. If you incorrectly specified `counter <`

`10` as the condition, the loop would iterate only nine times. This is a common *logic error* called an **off-by-one error**.

Common Programming Error 5.2

Using an incorrect relational operator or an incorrect final value of a loop counter in the loop-continuation condition of an iteration statement can cause an off-by-one error.

Error-Prevention Tip 5.2

Using the final value and operator `<=` *in a loop's condition helps avoid off-by-one errors. For a loop that outputs 1 to 10, the loop-continuation condition should be* `counter <= 10` *rather than* `counter < 10` *(which causes an off-by-one error) or* `counter < 11` *(which is correct). Many programmers prefer so-called zero-based counting, in which to count 10 times,* `counter` *would be initialized to zero and the loop-continuation test would be* `counter < 10`.

Error-Prevention Tip 5.3

As Chapter 4 mentioned, integers can overflow, causing logic *errors. A loop's control variable also could overflow. Write your loop conditions carefully to prevent this.*

A Closer Look at the `for` Statement's Header

Figure 5.3 takes a closer look at the `for` statement in Fig. 5.2. The first line—including the keyword `for` and everything in parentheses after `for` (line 8 in Fig. 5.2)—is sometimes called the `for` **statement header**. The `for` header "does it all"—it specifies each item needed for counter-controlled iteration with a control variable. If there's more than one statement in the body of the `for`, braces are required to define the body of the loop.

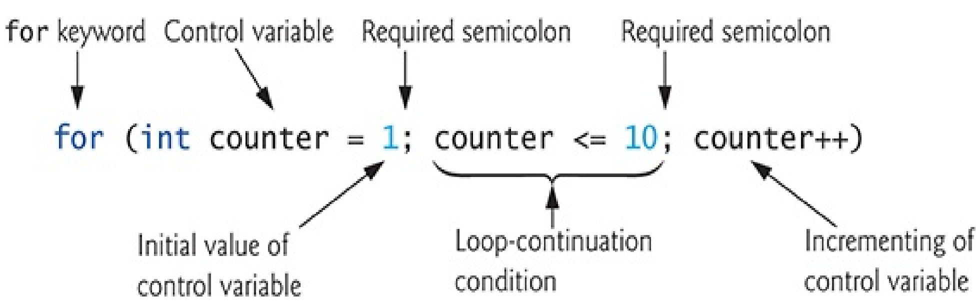


Fig. 5.3

`for` statement header components.

Description

General Format of a `for` Statement

The general format of the `for` statement is

```
for (initialization; loopContinuationCondition; incre
   statements
}
```

where the *initialization* expression names the loop's control variable and *optionally* provides its initial value, *loopContinuationCondition* determines whether the loop should continue executing and *increment* modifies the control variable's value, so that the loop-continuation condition eventually becomes `false`. The two semicolons in the `for` header are required. If the loop-continuation condition is initially `false`, the program does *not* execute the body. Instead, execution proceeds with the statement following the `for`.

Representing a `for` Statement with an Equivalent `while` Statement

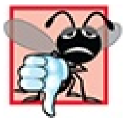
The `for` statement often can be represented with an equivalent `while` statement as follows:

```
initialization;
while (loopContinuationCondition) {
   statements
   increment;
}
```

In Section 5.8, we show a case in which a `for` statement cannot be represented with an equivalent `while` statement. Typically, `for` statements are used for counter-controlled iteration and `while` statements for sentinel-controlled iteration. However, `while` and `for` can each be used for either iteration type.

Scope of a `for` Statement's Control Variable

If the *initialization* expression in the `for` header declares the control variable (i.e., the control variable's type is specified before the variable name, as in Fig. 5.2), the control variable can be used *only* in that `for` statement—it will not exist outside it. This restricted use is known as the variable's **scope**. The scope of a variable defines where it can be used in a program. For example, a *local variable* can be used *only* in the method that declares it and *only* from the point of declaration through the next right brace (`}`), which is often the brace that closes the method body. Scope is discussed in detail in Chapter 6, Methods: A Deeper Look.

 Common Programming Error 5.3

When a `for` *statement's control variable is declared in the initialization section of the* `for`*'s header, using the control variable after the* `for`*'s body is a compilation error.*

Expressions in a `for` Statement's Header Are Optional

The three expressions in a `for` header are optional. If the *loopContinuationCondition* is omitted, Java assumes that it's *always* `true`, thus creating an *infinite loop*. You might omit the *initialization* expression if the program initializes the control variable *before* the loop. You might omit the *increment* expression if the program calculates the increment with statements in the loop's body or if no increment is needed. The increment expression in a `for` acts as if it were a standalone statement at the end of the `for`'s body. So, the expressions

```
counter = counter + 1
```

```
counter += 1
++counter
counter++
```

are equivalent increment expressions in a `for` statement. Many programmers prefer `counter++` because it's concise and because a `for` loop evaluates its increment expression *after* its body executes, so the postfix increment form seems more natural. In this case, the variable being incremented does not appear in a larger expression, so preincrementing and postincrementing actually have the *same* effect.

Common Programming Error 5.4

Placing a semicolon immediately to the right of the right parenthesis of a `for` *header makes that* `for`*'s body an empty statement. This is normally a logic error.*

Error-Prevention Tip 5.4

Infinite loops occur when the loop-continuation condition in an iteration statement never becomes `false`*. To prevent this situation in a counter-controlled loop, ensure that the control variable is modified during each iteration of the loop so that the loop-continuation condition will eventually become* `false`. *In a sentinel-controlled loop, ensure that the sentinel value is able to be input.*

Placing Arithmetic Expressions in a `for` Statement's Header

The initialization, loop-continuation condition and increment portions of a `for` statement can contain arithmetic expressions. For example, assume that `x = 2` and `y = 10`. If `x` and `y` are not modified in the body of the loop, the statement

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

is equivalent to the statement

```
for (int j = 2; j <= 80; j += 5)
```

The increment of a `for` statement may also be *negative*, in which case it's a **decrement**, and the loop counts *downward*.

Using a `for` Statement's

Control Variable in the Statement’s Body

Programs frequently display the control-variable value or use it in calculations in the loop body, but this use is *not* required. The control variable is commonly used to control iteration *without* being mentioned in the body of the `for`.

Error-Prevention Tip 5.5

Although the value of the control variable can be changed in the body of a `for` *loop, avoid doing so, because this practice can lead to subtle errors.*

UML Activity Diagram for the `for` Statement

The `for` statement’s UML activity diagram is similar to that of the `while` statement (Fig. 4.6). Figure 5.4 shows the activity diagram of the `for` statement in Fig. 5.2. The diagram makes it clear that initialization occurs *once before* the loop-continuation test is evaluated the first time, and that incrementing occurs *each* time through the loop *after* the body statement executes.

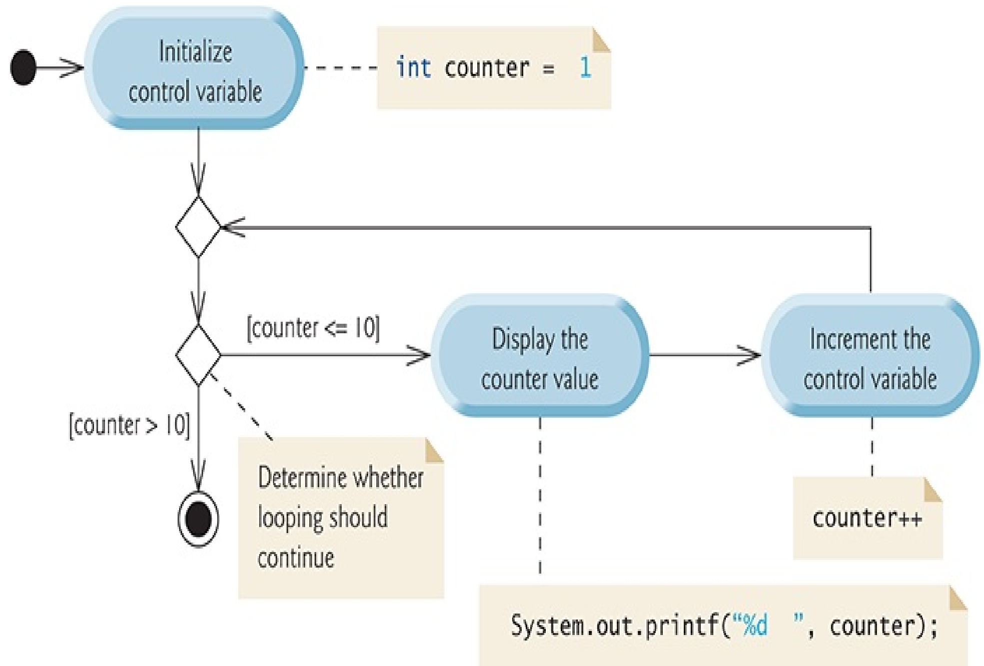


Fig. 5.4

UML activity diagram for the `for` statement in Fig. 5.2.

Description