# Programming via Java More on loops

In this chapter we look at two new categories of statements — the **`for`** loop and the **`break`** statement. Neither is very complex conceptually, but nonetheless they are convenient enough in practice that you wouldn't want to miss them.

## 9.1. The `for` loop

You may have noticed by now that our programs often have loops incorporating some sort of counter variable, as in the following.

```
int <var> = <initial>;
while(<var> < <maximum>) {
    // do something once
    var++;
}
```

This happens often enough that Java provides a shortcut for the situation in the form of a different type of loop called a **`for`** loop. The above example could instead be written as the following.

```
for(int <var> = <initial>; <var> < <maximum>; <var>++) {
    // do something once
}
```

Inside the parentheses of a **`for`** loop are three parts separated by semicolons.

```
for(<initialization>; <condition>; <update>) {
    <statementsToRepeat>
}
```

When the Java compiler sees this, it automatically interprets it as if the user had written the following instead.

```
<initialization>;
while(<condition>) {
    <statementsToRepeat>
    <update>;
}
```

In some sense, you may see the **`while`** loop as being easier to understand. But you should quickly be able to adapt to learning the **`for`** loop. Most programmers strongly prefer **`for`** loops to **`while`** loops whenever a program iterates over a sequence of numbers. This is because all the information about the numeric sequence is contained on one line, which makes the program shorter, easier to understand, and less liable to lead to bugs.

These aren't really exactly equivalent. A significant difference is that if ***`<initialization>`*** incorporates a variable declaration, the declared variable's scope would be restricted to only within the **`for`** loop, but in the equivalent **`while`** loop formulation, the variable could be accessed following the **`while`** loop's end.

(Another difference has to do with the **`continue`** statement, but this statement isn't often useful, and so we'll defer discussion of it to later.)

As an example of this in practice, suppose we wanted to modify `MovingBall` of Figure 6.3 so that the program displays only 50 frames of the ball's movement before the program ends. The program of Figure 9.1 manages to accomplish this. Notice how it introduces a **`for`** loop to count how many frames have been displayed thus far.

**Figure 9.1:** The `MovingBall50` program.

```
1  import acm.program.*;
2  import acm.graphics.*;
3  import java.awt.*;
4
5  public class MovingBall50 extends GraphicsProgram {
6      public void run() {
7          GOval ball = new GOval(25, 25, 50, 50);
8          ball.setFilled(true);
9          ball.setFillColor(new Color(255, 0, 0));
10         add(ball);
11
12         for(int frames = 0; frames < 50; frames++) {
13             pause(40);
14             ball.move(3, 2);
15         }
16     }
17 }
```

The **`for`** loop can be modified for virtually any numeric sequence. Suppose we want to count by fives up to 100.

```
for(int current = 5; current <= 100; current += 5) {
    println(current);
}
```

Here, we've modified the ***<update>*** clause so that the `current` variable goes up by 5 with each iteration. Notice also that the ***<condition>*** clause uses less-than-or-equal rather than less-than.

Suppose we want instead to count *down* from 10 to 1. In this case, we'll start our variable at 10, and we'll modify the ***<update>*** clause so that the variable is decremented with each iteration rather than incremented.

```
for(int current = 10; current >= 1; current--) {
    println(current);
}
```

# 9.2. The `break` statement

The **`break`** statement is another type of statement that sometimes turns out to be useful. It looks quite simple.

```
break;
```

When the computer reaches a **`break`** statement, it will immediate exit whichever loop it is in, proceeding to the statement following the loop's body. It's important to remember that an **`if`** statement is not a loop: If the **`break`** occurs in an **`if`** statement (as it almost always will be), the computer will look successive levels outside the **`if`**statement to find the loop from which to break.

The program of Figure 9.2 illustrates an example where one might use a **`break`** statement. Here, after each movement of the ball, we test to see whether the ball has gone off the the window's edge; if the ball has gone off, the computer executes a **`break`** statement, which moves execution to after the **`while`** loop. In this program's case, the ball would start moving backwards indefinitely.

**Figure 9.2:** The `MovingBallBreak` program.

```
1  import acm.program.*;
2  import acm.graphics.*;
3  import java.awt.*;
4
```

```
5  public class MovingBallBreak extends GraphicsProgram {
6      public void run() {
7          GOval ball = new GOval(25, 25, 50, 50);
8          ball.setFilled(true);
9          ball.setFillColor(new Color(255, 0, 0));
10         add(ball);
11
12         while(true) {
13             pause(40);
14             ball.move(3, 2);
15             if(ball.getX() > getWidth() || ball.getY() > getHeight()) {
16                 break;
17             }
18         }
19         while(true) {
20             pause(40);
21             ball.move(-3, -2);
22         }
23     }
24 }
```

(Incidentally, most programmers would argue that this program would be better written without a **`break`** statement. Instead, the **`if`** statement would go inside the**`while`** loop's condition. Still, this is a handy example with which to illustrate our point.)

Sometimes you'll have one loop inside another. If the computer encounters a **`break`** statement, it applies only to the innermost loop. In the below example, the ball will go back and forth across the window 10 times. Each time the computer encounters the first **`break`** statement, it will break out of the first **`while`** loop rather than the **`for`**loop, since the **`while`** loop is the innermost loop containing that statement; and it would continue on to the second **`while`** loop. And each time the computer encounters the second **`break`** statement, the computer will decide whether to repeat the **`for`** loop for another lap.

**Figure 9.3:** The `MovingBallLaps` program.

```
1  import acm.program.*;
2  import acm.graphics.*;
3  import java.awt.*;
4
5  public class MovingBallLaps extends GraphicsProgram {
6      public void run() {
```

```
7           GOval ball = new GOval(25, 25, 50, 50);
8           ball.setFilled(true);
9           ball.setFillColor(new Color(255, 0, 0));
10          add(ball);
11
12          for(int laps = 0; laps < 10; laps++) {
13              while(true) {
14                  pause(40);
15                  ball.move(3, 2);
16                  if(ball.getX() > getWidth() || ball.getY() > getHeight())
{
17                      break;
18                  }
19              }
20              while(true) {
21                  pause(40);
22                  ball.move(-3, -2);
23                  if(ball.getX() < -50 || ball.getY() < -50) {
24                      break;
25                  }
26              }
27          }
28      }
29  }
```

**Source: http://www.toves.org/books/java/ch09-moreloop/index.html**