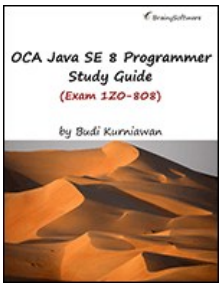

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Chapter 2: Statements

Overview

A computer program is a compilation of instructions called statements. There are many types of statements in Java and some—such as **if, while, for**, and **switch**—are conditional statements that determine the program flow. This chapter discusses Java statements, starting with an overview and then providing details of each of them. The **return** statement, which is the statement to exit a method, is discussed in Chapter 3, "Objects and Classes."

2.1 Overview

In programming, a statement is an instruction to do something. Statements control the sequence of program execution. Assigning a value to a variable is an example of a statement.

```
x = z + 5;
```

Even a variable declaration is a statement.

```
long secondsElapsed;
```

By contrast, an *expression* is a combination of operators and operands that gets evaluated. For example, **z + 5** is an expression.

In Java a statement is terminated with a semicolon and multiple statements can be written in a single line.

```
x = y + 1; z = y + 2;
```

However, writing multiple statements in a single line is not recommended as it obscures code readability.

Note In Java, an empty statement is legal and does nothing:

```
    ;
```

Some expressions can be made statements by terminating them with a semicolon. For example, **x++** is an expression. However, this is a statement:

```
x++;
```

Statements can be grouped in a block. By definition, a block is a sequence of the following programming elements within braces:

- statements
- local class declarations
- local variable declaration statements

A statement and a statement block can be labeled. Label names follow the same rule as Java identifiers and are terminated with a colon. For example, the following statement is labeled **sectionA**.

```
sectionA: x = y + 1;
```

And, here is an example of labeling a block:

```
start: {
    // statements
}
```

The purpose of labeling a statement or a block is so that it can be referenced by the **break** and **continue** statements.

2.2 The if Statement

The **if** statement is a conditional branch statement. The syntax of the **if** statement is either one of these two:

```
if (booleanExpression) {
    statement(s)
```

```
}

if (booleanExpression) {
    statement(s)
} else {
    statement(s)
}
```

If *booleanExpression* evaluates to **true**, the statements in the block following the **if** statement are executed. If it evaluates to **false**, the statements in the **if** block are not executed. If *booleanExpression* evaluates to **false** and there is an **else** block, the statements in the **else** block are executed.

For example, in the following **if** statement, the **if** block will be executed if **x** is greater than 4.

```
if (x > 4) {
    // statements
}
```

In the following example, the **if** block will be executed if **a** is greater than 3. Otherwise, the **else** block will be executed.

```
if (a > 3) {
    // statements
} else {
    // statements
}
```

Note that the good coding style suggests that statements in a block be indented.

If you are evaluating a boolean in your if statement, it's not necessary to use the == operator like this:

```
boolean fileExist = ...
if (fileExist == true) {
```

Instead, you can simply write

```
if (fileExists) {
```

By the same token, instead of writing

```
if (fileExists == false) {
```

write

```
if (!fileExists) {
```

If the expression to be evaluated is too long to be written in a single line, it is recommended that you use two units of indentation for subsequent lines. For example.

```
if (numberOfLoginAttempts < numberOfMaximumLoginAttempts
        || numberOfMinimumLoginAttempts > y) {
    y++;
}
```

If there is only one statement in an **if** or **else** block, the braces are optional.

```
if (a > 3)
    a++;
else
    a = 3;
```

However, this may pose what is called the dangling else problem. Consider the following example:

```
if (a > 0 || b < 5)
    if (a > 2)
        System.out.println("a > 2");
    else
        System.out.println("a < 2");
```

The **else** statement is dangling because it is not clear which **if** statement the **else** statement is associated with. An **else** statement is always associated with the immediately preceding **if**. Using braces makes your code clearer.

```
if (a > 0 || b < 5) {
```

```
    if (a > 2) {
        System.out.println("a > 2");
    } else {
        System.out.println("a < 2");
    }
}
```

If there are multiple selections, you can also use **if** with a series of **else** statements.

```
if (booleanExpression1) {
    // statements
} else if (booleanExpression2) {
    // statements
}
...
else {
    // statements
}
```

For example

```
if (a == 1) {
    System.out.println("one");
} else if (a == 2) {
    System.out.println("two");
} else if (a == 3) {
    System.out.println("three");
} else {
    System.out.println("invalid");
}
```

In this case, the **else** statements that are immediately followed by an **if** do not use braces. See also the discussion of the **switch** statement in the section "The switch Statement" later in this chapter.

2.3 The while Statement

In many occasions, you may want to perform an action several times in a row. In other words, you have a block of code that you want executed repeatedly. Intuitively, this can be done by repeating the lines of code. For instance, a beep can be achieved using this line of code:

```
java.awt.Toolkit.getDefaultToolkit().beep();
```

And, to wait for half a second you use these lines of code.

```
try {
    Thread.currentThread().sleep(500);
} catch (Exception e) {
}
```

Therefore, to produce three beeps with a 500 milliseconds interval between two beeps, you can simply repeat the same code:

```
java.awt.Toolkit.getDefaultToolkit().beep();
try {
    Thread.currentThread().sleep(500);
} catch (Exception e) {
}

java.awt.Toolkit.getDefaultToolkit().beep();
try {
    Thread.currentThread().sleep(500);
} catch (Exception e) {
}
java.awt.Toolkit.getDefaultToolkit().beep();
```

However, there are circumstances where repeating code does not work. Here are some of those:

- The number of repetition is higher than 5, which means the number of lines of code increases five fold. If there is a line that you need to fix in the block, copies of the same line must also be modified.
- If the number of repetitions is not known in advance.

A much cleverer way is to put the repeated code in a loop. This way, you only write the code once but you can instruct Java to

execute the code any number of times. One way to create a loop is by using the **while** statement, which is the topic of discussion of this section. Another way is to use the **for** statement, which is explained in the next section.

The **while** statement has the following syntax.

```
while (booleanExpression) {
    statement(s)
}
```

Here, *statement(s)* will be executed as long as *booleanExpression* evaluates to **true**. If there is only a single statement inside the braces, you may omit the braces. For clarity, however, you should always use braces even when there is only one statement.

As an example of the **while** statement, the following code prints integer numbers that are less than three.

```
int i = 0;
while (i < 3) {
    System.out.println(i);
    i++;
}
```

Note that the execution of the code in the loop is dependent on the value of **i**, which is incremented with each iteration until it reaches 3.

To produce three beeps with an interval of 500 milliseconds, use this code:

```
int j = 0;
while (j < 3) {
    java.awt.Toolkit.getDefaultToolkit().beep();
    try {
        Thread.currentThread().sleep(500);
    } catch (Exception e) {
    }
    j++;
}
```

Sometimes, you use an expression that always evaluates to **true** (such as the **boolean** literal **true)** but relies on the **break** statement to escape from the loop.

```
int k = 0;
while (true) {
    System.out.println(k);
    k++;
    if (k > 2) {
        break;
    }
}
```

You will learn about the **break** statement in the section, "The break Statement" later in this chapter.

2.4 The do-while Statement

The **do-while** statement is like the **while** statement, except that the associated block always gets executed at least once. Its syntax is as follows:

```
do {
    statement(s)
} while (booleanExpression);
```

With **do-while**, you put the statement(s) to be executed after the **do** keyword. Just like the **while** statement, you can omit the braces if there is only one statement within them. However, always use braces for the sake of clarity.

For example, here is an example of the **do-while** statement:

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 3);
```

This prints the following to the console:

```
0
1
2
```

The following **do-while** demonstrates that at least the code in the **do** block will be executed once even though the initial value of **j** used to test the expression **j < 3** evaluates to **false**.

```
int j = 4;
do {
    System.out.println(j);
    j++;
} while (j < 3);
```

This prints the following on the console.

```
4
```

2.5 The for Statement

The **for** statement is like the **while** statement, i.e. you use it to enclose code that needs to be executed multiple times. However, **for** is more complex than **while**.

The **for** statement starts with an initialization, followed by an expression evaluation for each iteration and the execution of a statement block if the expression evaluates to **true**. An update statement will also be executed after the execution of the statement block for each iteration.

The **for** statement has following syntax:

```
for ( init ; booleanExpression ; update ) {
    statement(s)
}
```

Here, *init* is an initialization that will be performed before the first iteration, *booleanExpression* is a boolean expression which will cause the execution of *statement(s)* if it evaluates to **true**, and *update* is a statement that will be executed *after* the execution of the statement block. *init, expression*, and *update* are optional.

The **for** statement will stop only if one of the following conditions is met:

- *booleanEpression* evaluates to **false**
- A **break** or **continue** statement is executed
- A runtime error occurs.

It is common to declare a variable and assign a value to it in the initialization part. The variable declared will be visible to the *expression* and *update* parts as well as to the statement block.

For example, the following **for** statement loops three times and each time prints the value of **i**.

```
for (int i = 0; i < 3; i++) {
    System.out.println(i);
}
```

The **for** statement starts by declaring an **int** named **i** and assigning 0 to it:

```
int i = 0;
```

It then evaluates the expression **i < 3**, which evaluates to **true** since **i** equals 0. As a result, the statement block is executed, and the value of **i** is printed. It then performs the update statement **i++**, which increments **i** to 1. That concludes the first loop.

The **for** statement then evaluates the value of **i < 3** again. The result is again **true** because **i** equals 1. This causes the statement block to be executed and **1** is printed on the console. Afterwards, the update statement **i++** is executed, incrementing **i** to 2. That concludes the third loop.

Next, the expression **i < 3** is evaluated and the result is **true** because **i** equals 2. This causes the statement block to be run and 2 is printed on the console. Afterwards, the update statement **i++** is executed, causing **i** to be equal to 3. This concludes

the second loop.

Next, the expression **i < 3** is evaluated again, and the result is **false**. This stops the **for** loop.

This is what you see on the console:

```
0
1
2
```

Note that the variable **i** is not visible anywhere else since it is declared within the **for** loop.

Note also that if the statement block within **for** only consists of one statement, you can remove the braces, so in this case the above **for** statement can be rewritten as:

```
for (int i = 0; i < 3; i++)
    System.out.println(i);
```

However, using braces even if there is only one statement makes your code clearer.

Here is another example of the **for** statement.

```
for (int i = 0; i < 3; i++) {
    if (i % 2 == 0) {
        System.out.println(i);
    }
}
```

This one loops three times. For each iteration the value of **i** is tested. If **i** is even, its value is printed. The result of the **for** loop is as follows:

```
0
2
```

The following **for** loop is similar to the previous case, but uses **i += 2** as the update statement. As a result, it only loops twice, when **i** equals 0 and when it is 2.

```
for (int i = 0; i < 3; i += 2) {
    System.out.println(i);
}
```

The result is

```
0
2
```

A statement that decrements a variable is often used too. Consider the following **for** loop:

```
for (int i = 3; i > 0; i--) {
    System.out.println(i);
}
```

which prints:

```
3
2
1
```

The initialization part of the **for** statement is optional. In the following **for** loop, the variable **j** is declared outside the loop, so potentially **j** can be used from other points in the code outside the **for** statement block.

```
int j = 0;
for ( ; j < 3; j++) {
    System.out.println(j);
}
// j is visible here
```

As mentioned previously, the update statement is optional. The following **for** statement moves the update statement to the end of the statement block. The result is the same.

```
int k = 0;
```

```
for ( ; k < 3; ) {
    System.out.println(k);
    k++;
}
```

In theory, you can even omit the *booleanExpression* part. For example, the following **for** statement does not have one, and the loop is only terminated with the **break** statement. See the section, "The break Statement" for more information.

```
int m = 0;
for ( ; ; ) {
    System.out.println(m);
    m++;
    if (m > 4) {
        break;
    }
}
```

If you compare **for** and **while**, you'll see that you can always replace the **while** statement with **for**. This is to say that

```
while (expression) {
    ...
}
```

can always be written as

```
for ( ; expression; ) {
    ...
}
```

Note In addition, **for** can iterate over an array or a collection. See Chapters 5, "Arrays" for a discussion of the enhanced **for**.

2.6 The break Statement

The **break** statement is used to break from an enclosing **do, while, for**, or **switch** statement. It is a compile error to use **break** anywhere else.

For example, consider the following code

```
int i = 0;
while (true) {
    System.out.println(i);
    i++;
    if (i > 3) {
        break;
    }
}
```

The result is

```
0
1
2
3
```

Note that **break** breaks the loop without executing the rest of the statements in the block.

Here is another example of break, this time in a **for** loop.

```
int m = 0;
for ( ; ; ) {
    System.out.println(m);
    m++;
    if (m > 4) {
        break;
    }
}
```

The **break** statement can be followed by a label. The presence of a label will transfer control to the start of the code identified by the label. For example, consider this code.

```
start:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
```

```
        if (j == 2) {
            break start;
        }
        System.out.println(i + ":" + j);
    }
}
```

The use of label start identifies the first **for** loop. The statement **break start;** therefore breaks from the first loop. The result of running the preceding code is as follows.

```
0:0
0:1
```

Java does not have a goto statement like in C or C++, and labels are meant as a form of goto. However, just as using goto in C/C++ may obscure your code, the use of labels in Java may make your code unstructured. The general advice is to avoid labels if possible and to always use them with caution.

2.7 The continue Statement

The **continue** statement is like **break** but it only stops the execution of the current iteration and causes control to begin with the next iteration.

For example, the following code prints the numbers 0 to 9, except 5.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue;
    }
    System.out.println(i);
}
```

When **i** is equals to 5, the expression of the **if** statement evaluates to **true** and causes the **continue** statement to be called. As a result, the statement below it that prints the value of **i** is not executed and control continues with the next loop, i.e. for **i** equal to 6.

As with **break, continue** may be followed by a label to identify which enclosing loop to continue to. As with labels with **break**, employ **continue label** with caution and avoid it if you can.

Here is an example of **continue** with a label.

```
start:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        if (j == 2) {
            continue start;
        }
        System.out.println(i + ":" + j);
    }
}
```

The result of running this code is as follows:

```
0:0
0:1
1:0
1:1
2:0
2:1
```

2.8 The switch Statement

The **switch** statement is an alternative to a series of **else if. switch** allows you to choose a block of statements to run from a selection of code, based on the return value of an expression. The expression used in the **switch** statement must return an **int**, a **String**, or an enumerated value.

Note The **String** class is discussed in Chapter 4, "Core Classes."

The syntax of the **switch** statement is as follows.

```
switch(expression) {
```

```
case value_1 :
    statement(s);
    break;
case value_2 :
    statement(s);
    break;
  .
  .
  .
case value_n :
    statement(s);
    break;
default:
    statement(s);
}
```

Failure to add a **break** statement after a case will not generate a compile error but may have more serious consequences because the statements on the next case will be executed.

Here is an example of the **switch** statement. If the value of **i** is 1, "One player is playing this game." is printed. If the value is 2, "Two players are playing this game is printed." If the value is 3, "Three players are playing this game is printed. For any other value, "You did not enter a valid value." will be printed.

```
int i = ...;
switch (i) {
case 1 :
    System.out.println("One player is playing this game.");
    break;
case 2 :
    System.out.println("Two players are playing this game.");
    break;
case 3 :
    System.out.println("Three players are playing this game.");
    break;
default:
    System.out.println("You did not enter a valid value.");
}
```

For examples of switching on a String or an enumerated value, see Chapter 4, "Core Classes."

Self Test

1 Question ?

Given

```
int j = 0;
for (int i = 0; i < 3; i++) {
    if (i % 2 == 0) j = i;
}
System.out.println(j);
```

What is the output of this code?

A. 0

B. 1

C. 2

D. 3

2 Question ?

Consider this code snippet:

```
int y = 0;
for (int x = 0; y < 5; ++x) {
    if (x % 2 == 0) continue;
    y += x;
}
System.out.println(y);
```

What is the output of this code?

A. 3

B. 5

C. 7

D. 9

3 Question ?

Given

```
int y = 0;
for (; ; ) {
    if (y >= 10) break;
    y += ++y;
}
System.out.println(y);
```

What is the output of this code?

A. 14

B. 15

C. 16

D. None of the above

4 Question ?

Given

```
int y = 0;
for (; ; ) {
    if (y >= 10) break;
    y += y++;
}
System.out.println(y);
```

What is the output of this code?

A. 14

B. 15

C. 16

D. None of the above

5 Question ?

Given

```
int i = 1;
switch (i) {
case 1 :
    System.out.println("One player is playing this game.");
case 2 :
    System.out.println("Two players are playing this game.");
    break;
default:
    System.out.println("You did not enter a valid value.");
}
```

What is printed on the console if the code is executed?

A. One player is playing this game.

B. Two players are playing this game.

C. One player is playing this game.
Two players are playing this game.

D. None of the above.

6 Question ?

Given

```
int i = 1;
int y = 5;
if (i == 2)
if (i == 1)
    y = 50;
else y = 500;
```

What is the value of y after the code is executed?

A. 5.

B. 50.

C. 500.

D. None of the above.

7 Question ?

Given this code

```
1.  ...
2.      System.out.print(i + " ");
3.  }
```

and given the following result

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

What line of code should be inserted into line 1?

A. for (int i = 0; i < 10; i++) {

B. for (int i = 10; i > 0; i--) {

C. while (i < 10) {

D. while (i > 10) {

8 Question ?

Consider the following code.

```
1.   int x = 0;
2.   while (x < 10) {
3.       x++;
4.       System.out.print(x + " ");
5.   }
```

What is the equivalent for statement that can be used to replace lines 1 to 3?

A. for (int i = 1; i <= 10; i++) {

B. for (int i = 1; i < 11; i++) {

C. for (int i = 1; i < 11; ++i) {

D. for (int i = 1; i <= 10; ++i) {

9 Question ?

This code prints all odd numbers between 0 and 10.

```
1.  for (int i = 1; i <= 10; i++) {
2.      if (i % 2 == 1)
3.          System.out.print(i + " ");
4.  }
```

Which of the following for statements can be used to replace lines 1 and 2?

A. for (int i = 1; i <= 10; i++) {

B. for (int i = 1; i <= 10; i+=2) {

C. for (int i = 1; i <= 10; i--) {

D. None of the above

10 Question

Given

```
int i = 1;
do {
    System.out.print(i++);
} while (i != 5);
```

What is the output of the program?

A. 012345

B. 01234

C. 1234

D. 12345

Answers

1 **B**

The code in the **if** statement is only executed when **i** is evenly divisible by two, i.e. when i equals 0 or 2. The last statement is j = i where i = 2, therefore the final value of **j** is 2.

2 **D**

y is incremented when **x** is odd (i.e. when **i** equals 1, 3, 5, …). In addition, the for loop will only continue until y is equal to or less than 5. Therefore, y = 1 when x = 1, y = 4 when x = 3, and y = 9 when x = 5. When y reaches 9, the loop stops as the condition (**y** < **5**) is no longer met.

3 **B**

The for statement is an infinite loop that will exit only if y is equal to or greater than 10. When the increment operator ++ is used in prefix notation (e.g. ++y), the operand is incremented first and then its value is used in an expression. The statement y += ++y is equivalent to y = y + ++y.

At iteration 1, the value of y at the beginning of the loop is 0. The statement becomes

```
y = 0 + ++y
```

The operand y is incremented and its value used in the expression, thus y = 0 + 1 and y is equal to 1.

At iteration 2, the value of y at the beginning of the loop is 1. The statement becomes

```
y = 1 + ++y = 1 + 2 = 3
```

At iteration 3, the value of y at the beginning of the loop is 3. The statement becomes

```
y = 3 + ++y = 3 + 4 = 7
```

At iteration 4, the value of y at the beginning of the loop is 7. The statement becomes

```
y = 7 + ++y = 7 + 8 = 15
```

4 **D**.

The **for** statement is an infinite loop that will exit only if **y** is equal to or greater than 10. When the increment operator ++ is used in postfix notation (e.g. y++), the value of the operand is used in an expression and then it is incremented. The statement y += y++ is equivalent to y = y + y++.

At iteration 1, the value of y at the beginning of the loop is 0. The statement becomes

```
y = 0 + y++
```

The value of y is 0 and is used in the expression 0 + 0. y is then incremented by one but the y on the left of the equal sign is assigned the result of evaluating the expression and its value is therefore 0.

At iteration 2, the value of y is again 0 and the **for** loop goes on indefinitely.

5 **C**.

Because there is no break statement after the first case, the execution flows through the second case. Hence, C.

6 **A**.

This is the dangling else problem. It is not clear which if statement the else statement is associated with. That's why it is better to use curly

brackets with if. In this case, the else statement belongs to the if statement closest to it. The code is the same as this.

```
int i = 1;
int y = 5;
if (i == 2) {
    if (i == 1) {
        y = 50;
    } else {
        y = 500;
    }
}
```

It is now clear that y is never re-assigned a new value.

7 **A**.

Obviously, you need a loop that starts i at 10 and decrements it by one until it reaches 1.

8 **A, B, C, D**.

Any of the for statements can be used to replace the while statement.

9 **B**.

The update statement **i+=2** in B increments i by 2.

10 **C**.

The **do** block is executed four times, from when i = 1 to i = **4**.