JB0230: Java OOP: Flow of Control*

R.G. (Dick) Baldwin

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0†

Abstract

Java supports several different statements designed to alter or control the logical flow of the program. This module explores those statements.

1 Table of Contents

- Preface (p. 1)
  - Viewing tip (p. 2)
    - Figures (p. 2)
    - Listings (p. 2)
- Introduction (p. 2)
  - Flow of control (p. 2)
  - The while statement (p. 3)
  - The if-else statement (p. 5)
  - The switch-case statement (p. 5)
  - The for loop (p. 6)
  - The for-each loop (p. 10)
  - The do-while loop (p. 10)
  - The break and continue statements (p. 11)
  - Unlabeled break and continue (p. 11)
  - Labeled break and continue statements (p. 11)
    - Labeled break statements (p. 12)
    - Labeled continue statements (p. 14)
  - The return statement (p. 14)
  - Exception handling (p. 16)
- Looking ahead (p. 16)
- Miscellaneous (p. 16)

2 Preface

Java supports several different statements designed to alter or control the logical flow of the program. This module explores those statements.

*Version 1.3: Nov 25, 2012 10:38 pm +0000
†http://creativecommons.org/licenses/by/3.0/

2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

2.1.1 Figures

- Figure 1 (p. 3) . Statements that support flow of control.
- Figure 2 (p. 4) . Syntax of a while statement.
- Figure 3 (p. 5) . Syntax of an if-else statement.
- Figure 4 (p. 6) . Syntax of a switch-case statement.
- Figure 5 (p. 7) . Syntax of a for loop.
- Figure 6 (p. 11) . Syntax of a do-while loop.
- Figure 7 (p. 12) . Syntax of a labeled statement.
- Figure 8 (p. 15) . An empty return statement.
- Figure 9 (p. 15) . Returning a value from a method.

2.1.2 Listings

- Listing 1 (p. 4) . Sample Java while statement.
- Listing 2 (p. 8) . A program that won't compile.
- Listing 3 (p. 8) . Another program that won't compile.
- Listing 4 (p. 9) . A program that will compile.
- Listing 5 (p. 9) . Another program that will compile.
- Listing 6 (p. 12) . The program named switch1.java.
- Listing 7 (p. 13) . The program named switch2.java.

3 Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, types, expressions, flow-of-control, etc. This module concentrates on *flow-of-control* .

4 Flow of control

What is flow of control?

Java supports several different kinds of statements designed to alter or control the logical flow of the program.

The ability to alter the logical flow of the program is often referred to as *Flow of Control* .

Statements that support flow of control

Figure 1 (p. 3) lists the statements supported by Java for controlling the logical flow of the program.

Statements that support flow of control.

```
Statement            Type

if-else              selection
switch-case          selection
for                  loop
for-each             loop
while                loop
do-while             loop
try-catch-finally    exception handling
throw                exception handling
break                miscellaneous
continue             miscellaneous
label:               miscellaneous
return               miscellaneous
goto                 reserved by Java but not supported
```

Figure 1: Statements that support flow of control.

4.1 The while statement

We've seen the **while** statement in earlier modules. Several of the programs in earlier modules contained a **while** statement designed to control the logical flow of the program.

Syntax of a while statement

The general syntax of a **while** statement is shown in Figure 2 (p. 4) .

Syntax of a while statement.

```
while (conditional expression)
  statement or compound statement;
```

Figure 2: Syntax of a while statement.

Behavior of a while statement

The **three pillars** of procedural programming are

- sequence
- selection
- loop

The **while** statement is commonly used to create a loop structure, often referred to as a *while loop* .

Once the **while** statement is encountered in the sequence of code, the program will continue to execute the statement or compound statement shown in Figure 2 (p. 4) for as long as the conditional expression evaluates to true. *(Note that a compound statement is created by enclosing two or more statements inside a pair of matching curly brackets, thus creating a block of code as the body of the* ***while*** *statement or loop.)*

Sample Java *while* statement

The **while** statement shown in Listing 1 (p. 4) was extracted from a Java program in an earlier module.

Listing 1: Sample Java while statement.

```
while( (ch1 = System.in.read() ) != '#')
  ch2 = ch1;
```

The *in* variable of the *System* class

The **System** class defines a *class* variable named **in** . Because it is a *class* variable, it can be accessed using the name of the **System** class without the requirement to instantiate an object of the **System** class.

What the *in* variable contains

The **in** variable refers to an instance of a class that provides a **read** method that returns a character from the standard input device *(typically the keyboard)* .

Therefore, the expression **System.in.read()** in Listing 1 (p. 4) constitutes a call to the **read** method of the object referred to by the **in** variable of the **System** class.

A *while* loop is an entry condition loop

The **while** statement is used to form an *entry condition* loop. The significance of an entry condition loop is that the conditional expression is tested before the statements in the loop are executed. If it tests false initially, the statements in the loop are never executed.

The **while** loop shown in Listing 1 (p. 4) will continue reading characters from the keyboard for as long as the character entered is not the # character. *(Recall the not equal (!=) operator from an earlier module.)*

4.2 The if-else statement

The general syntax of an **if-else** statement is shown in Figure 3 (p. 5) .

Syntax of an if-else statement.

```
if(conditional expression)
  statement or compound statement;
else //optional
  statement or compound statement; //optional
```

Figure 3: Syntax of an if-else statement.

The **if-else** statement is the most basic of the statements used to control the logical flow of a Java program. It is used to satisfy the *selection* pillar mentioned earlier (p. 4) .

This statement will execute a specified block of code if some particular condition is true, and optionally, will execute a different block of code if the condition is not true.

The **else** clause shown in Figure 3 (p. 5) is optional. If it is not provided and the condition is not true, control simply passes to the next statement following the **If** statement with none of the code in the body of the **if** statement being executed. If the condition is true, the code in the body of the **if** statement is executed.

If the **else** clause is provided and the condition is true, the code in the body of the **if** clause is executed and the code in the body of the **else** clause is ignored.

If the **else** clause is provided and the condition is false, the code in the body of the **if** clause is ignored and the code in the body of the **else** clause is executed.

In all cases, control passes to the next statement following the **if-else** statement when the code in the **if-else** statement has finished executing. In other words, this is not a loop structure.

4.3 The switch-case statement

The **switch-case** statement is another implementation of the *selection* pillar mentioned earlier (p. 4) . The general syntax of a **switch-case** statement is shown in Figure 4 (p. 6) .

Syntax of a switch-case statement.

```
switch(expression){
  case constant:
    //sequence of optional statements
    break; //optional
  case constant:
    //sequence of optional statements
    break; //optional
.
.
.
  default //optional
    //sequence of optional statements
}
```

Figure 4: Syntax of a switch-case statement.

The type of the *expression*

According to the book, *Java Language Reference* , by Mark Grand, the expression shown in the first line in Figure 4 (p. 6) must be of type **byte** , **char** , **short** , or **int** .

The behavior of the switch-case statement

The expression is tested against a series of *case* constants of the same type as the expression. If a match is found, the sequence of optional statements associated with that *case* is executed.

Execution of statements continues until the optional **break** is encountered. When **break** is encountered, execution of the switch statement is terminated and control passes to the next statement following the switch statement.

If there is no **break** statement, all of the statements following the matching case will be executed including those in cases further down the page.

The optional default keyword

If no match is found and the optional default keyword along with a sequence of optional statements has been provided, those statements will be executed.

Labeled break

Java also supports labeled break statements. This capability can be used to cause Java to exhibit different behavior when switch statements are nested. This will be explained more fully in a later section on labeled break statements.

4.4 The for loop

The **for** statement is another implementation of the *loop* pillar mentioned earlier (p. 4) .

Actions of a *for loop*

The operation of a loop normally involves three actions in addition to executing the code in the body of the loop:

- Initialize a control variable.
- Test the control variable in a conditional expression.
- Update the control variable.

Grouping the actions

Java provides the **for** loop construct that groups these three actions in one place.

The syntax of a for loop

A **for** loop consists of three clauses separated by semicolons as shown in Figure 5 (p. 7) .

Syntax of a for loop.

```
for (first clause; second clause; third clause)
  single or compound statement
```

Figure 5: Syntax of a for loop.

Contents of the clauses

The first and third clauses can contain one or more expressions, separated by the *comma operator* .

The *comma operator*

The comma operator guarantees that its left operand will be executed before its right operand.

(While the comma operator has other uses in C++, this is the only use of the comma operator in Java.)

Behavior and purpose of the first clause

The expressions in the first clause are executed only once, at the beginning of the loop. Any legal expression(s) may be contained in the first clause, but typically the first clause is used for initialization.

Declaring and initializing variables in the first clause

Variables can be declared and initialized in the first clause, and this has an interesting ramification regarding scope that will be discussed later.

Behavior of the second clause

The second clause consists of a single expression that must evaluate to a **boolean** type with a value of true or false. The expression in the second clause must eventually evaluate to false to cause the loop to terminate.

Typically relational expressions or relational and conditional expressions are used in the second clause.

When the test is performed

The value of the second clause is tested when the statement first begins execution, and at the beginning of each iteration thereafter. Therefore, just like the **while** loop, the **for** loop is an *entry condition loop* .

When the third clause is executed

Although the third clause appears physically at the top of the loop, it isn't executed until the statements in the body of the loop have completed execution.

This is an important point since this clause is typically used to update the control variable, and perhaps other variables as well.

What the third clause can contain

Multiple expressions can appear in the third clause, separated by the comma operator. Again, those expressions will be executed from left to right. If variables are updated in the third clause and used in the body of the loop, it is important to understand that they do not get updated until the execution of the body is completed.

Declaring a variable in a *for* loop

As mentioned earlier, it is allowable to declare variables in the first clause of a **for** loop.

You can declare a variable with a given name outside *(prior to)* the **for** loop, or you can declare it inside the **for** loop, but not both.

If you declare it outside the **for** loop, you can access it either outside or inside the loop.

If you declare it inside the loop, you can access it only inside the loop. In other words, the scope of variables declared inside a **for** loop is limited to the loop.

This is illustrated in following sequence of four simple programs.

This program won't compile

The Java program shown in Listing 2 (p. 8) refuses to compile with a complaint that a variable named **cnt** has already been declared in the method when the attempt is made to declare it in the **for** loop.

Listing 2: A program that won't compile.

```
/*File for1.java Copyright 1997, R.G.Baldwin
This program will not compile because the variable
named cnt is declared twice.
*******************************************************/
class for1 { //define the controlling class
  public static void main(String[] args){ //main method
    int cnt = 5; //declare local method variable
    System.out.println(
          "Value of method var named cnt is " + cnt);

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
              "Value of loop var named cnt is " + cnt);

    System.out.println(
            "Value of method var named cnt is " + cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

The program shown in Listing 3 (p. 8) also won't compile, but for a different reason.

Listing 3: Another program that won't compile.

```
/*File for2.java Copyright 1997, R.G.Baldwin
This program will not compile because the variable
declared inside the for loop is not accessible
outside the loop.
*******************************************************/
```

```
class for2 { //define the controlling class
  public static void main(String[] args){ //main method

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
                  "Value of loop var named cnt is " + cnt);

    System.out.println(
                "Value of method var named cnt is " + cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

The declaration of the variable named **cnt** , outside the **for** loop, was removed from Listing 3 (p. 8) and the declaration inside the loop was allowed to remain. This eliminated the problem of attempting to declare the variable twice.

However, this program refused to compile because an attempt was made to access the variable named **cnt** outside the for loop. This was not allowed because the variable was declared inside the **for** loop and the scope of the variable was limited to the loop.

This program will compile

The Java program shown in Listing 4 (p. 9) will compile and run because the variable named **cnt** that is declared inside the **for** loop is accessed only inside the **for** loop. No reference to a variable with the same name appears outside the loop.

Listing 4: A program that will compile.

```
/*File for3.java Copyright 1997, R.G.Baldwin
This program will compile because the variable declared
inside the for loop is accessed only inside the loop.
************************************************************/
class for3 { //define the controlling class
  public static void main(String[] args){ //main method

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
                  "Value of loop var named cnt is " + cnt);
  }//end main
}//End controlling class.
```

This program will also compile

Similarly, the program shown in Listing 5 (p. 9) will compile and run because the variable named **cnt** was declared outside the **for** loop and was not declared inside the **for** loop. This made it possible to access that variable both inside and outside the loop.

Listing 5: Another program that will compile.

```
/*File for4.java Copyright 1997, R.G.Baldwin
This program will compile and run because the variable
named cnt is declared outside the for loop and is not
declared inside the for loop.
```

```
*********************************************************/
class for4 { //define the controlling class
  public static void main(String[] args){ //main method
    int cnt = 5; //declare local method variable
    System.out.println(
              "Value of method var named cnt is " + cnt);

    for(cnt = 0; cnt < 2; cnt++)
      System.out.println(
                  "Value of loop var named cnt is " + cnt);

    System.out.println(
                "Value of method var named cnt is " + cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

Empty clauses in a *for* loop

The first and third clauses in a **for** loop can be left empty but the semicolons must be there as placeholders.

One author suggests that even the middle clause can be empty, but it isn't obvious to this author how the loop would ever terminate if there is no conditional expression to be evaluated. Perhaps the loop could be terminated by using a break inside the loop, but in that case, you might just as well use a **while** loop.

4.5 The for-each loop

There is another form of loop structure that is often referred to as a **for-each** loop. In order to appreciate the benefits of this loop structure, you need to be familiar with Java collections and iterators, both of which are beyond the scope of this module.

As near as I can tell, there is nothing that you can do with the **for-each** loop that you cannot also do with the conventional **for** loop described above. Therefore, I rarely use it. You can find a description of the **for-each** loop on this Oracle website [1] .

I don't plan to discuss it further in this module. However, before you go for a job interview, you should probably do some online research and learn about it because an interviewer could use a question about the **for-each** loop to trip you up in the Q and A portion of the interview.

4.6 The do-while loop

The **do-while** loop is another implementation of the *loop* pillar mentioned earlier (p. 4) . However, it differs from the **while** loop and the **for** loop in one important respect; it is an *exit-condition* loop.

An exit-condition loop

Java provides an *exit-condition* loop having the syntax shown in Figure 6 (p. 11) .

[1] http://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html

Syntax of a do-while loop.

```
do {
    statements
              } while (conditional expression);
```

Figure 6: Syntax of a do-while loop.

Behavior

The statements in the body of the loop continue to be executed for as long as the conditional expression evaluates to true. An exit-condition loop guarantees that the body of the loop will be executed at least one time, even if the conditional expression evaluates to false the first time it is tested.

4.7 The break and continue statements

General behavior

Although some authors suggest that the **break** and **continue** statements provide an alternative to the infamous **goto** statement of earlier programming languages, it appears that the behaviors of the **labeled break** and **labeled continue** statements are much more restrictive than a general **goto** .

4.8 Unlabeled break and continue

The **break** and **continue** statements are supported in both labeled and unlabeled form.

First consider the behavior of break and continue in their unlabeled configuration.

Use of a *break* statement

The **break** statement can be used in a switch statement or in a loop. When encountered in a switch statement, break causes control to be passed to the next statement outside the innermost enclosing switch statement.

When break is encountered in a loop, it causes control to be passed to the next statement outside the innermost enclosing loop.

As you will see later, labeled break statements can be used to pass control to the next statement following switch or loop statements beyond the innermost switch or loop statement when those statements are nested.

Use of a continue statement

The continue statement cannot be used in a switch statement, but can be used inside a loop.

When an unlabeled continue statement is encountered, it causes the current iteration of the current loop to be terminated and the next iteration to begin.

A labeled continue statement can cause control to be passed to the next iteration of an outer enclosing loop in a nested loop situation.

An example of the use of an unlabeled switch statement is given in the next section.

4.9 Labeled break and continue statements

This section discusses the use of labeled break and continue statements.

4.9.1 Labeled break Statements

One way to describe the behavior of a labeled break in Java is to say: "Break all the way out of the labeled statement."

Syntax of a labeled statement

To begin with, the syntax of a labeled statement is a label followed by a colon ahead of the statement as shown in Figure 7 (p. 12) .

Syntax of a labeled statement.

```
myLabel:  myStatement;
```

Figure 7: Syntax of a labeled statement.

The label can be any legal Java identifier.

Behavior of labeled break

The behavior of a labeled break can best be illustrated using nested switch statements. For a comparison of labeled and unlabeled switch statements, consider the program shown in Listing 6 (p. 12) named **switch1** , which does not use a labeled break. Even though this program has a labeled statement, that statement is not referenced by a **break** . Therefore, the label is of no consequence.

Listing 6: The program named switch1.java.

```
/*File switch1.java
This is a Java application which serves as a baseline
comparison for switch2.java which uses a labeled break.
Note that the program uses nested switch statements.

The program displays the following output:

Match and break from here
Case 6 in outer switch
Default in outer switch
Beyond switch statements

*******************************************************/
class switch1 { //define the controlling class
  public static void main(String[] args){ //main method

  //Note that the following labeled switch statement is
  // not referenced by a labeled break in this program.
```

```
  // It will be referenced in the next program.
  outerSwitch: switch(5){//labeled outer switch statement
    case 5: //execute the following switch statement
    //Note that the code for this case is not followed
    // by break. Therefore, execution will fall through
    // the case 6 and the default.
      switch(1){ //inner switch statement
        case 1: System.out.println(
                            "Match and break from here");
                break; //break with no label
        case 2: System.out.println(
                           "No match for this constant");
                break;
      }//end inner switch statement

    case 6: System.out.println("Case 6 in outer switch");
    default: System.out.println(
                              "Default in outer switch");
  }//end outer switch statement

  System.out.println("Beyond switch statements");
  }//end main
}//End switch1 class.
```

After reviewing **switch1.java** , consider the same program named **switch2.java** shown in Listing 7 (p. 13) , which was modified to use a labeled break.

The outputs from both programs are shown in the comments at the beginning of the program. By examining the second program, and comparing the output from the second program with the first program, you should be able to see how the use of the labeled break statement causes control to break all the way out of the labeled switch statement.

Listing 7: The program named switch2.java.

```
/*File switch2.java
This is a Java application which uses a labeled break.
Note that the program uses nested switch statements.

See switch1.java for a comparison program which does not
use a labeled break.

The program displays the following output:

Match and break from here
Beyond switch statements
*********************************************************/
class switch2 { //define the controlling class
  public static void main(String[] args){ //main method

  outerSwitch: switch(5){//labeled outer switch statement
    case 5: //execute the following switch statement
```

```
    //Note that the code for this case is not followed by
    // break. Therefore, except for the labeled break at
    // case 1, execution would fall through the case 6 and
    // the default as demonstrated in the program named
    // switch1. However, the use of the labeled break
    // causes control to break all the way out of the
    // labeled switch bypassing case 6 and the default.
      switch(1){ //inner switch statement
        case 1: System.out.println(
                              "Match and break from here");
                break outerSwitch; //break with label
        case 2: System.out.println(
                             "No match for this constant");
                break;
      }//end inner switch statement

    case 6: System.out.println(
                                 "Case 6 in outer switch");
    default: System.out.println("Default in outer switch");
  }//end outer switch statement

  System.out.println("Beyond switch statements");
  }//end main
}//End switch1 class.
```

The modified program in Listing 7 (p. 13) uses a labeled break statement in the code group for *case 1* whereas the original program in Listing 6 (p. 12) has an unlabeled break in that position.

By comparing the output from this program with the output from the previous program, you can see that execution of the labeled break statement caused control to break all the way out of the labeled switch statement completely bypassing *case 6* and default.

As you can see from examining the output, the labeled break statement causes the program to break all the way out of the switch statement which bears a matching label.

A similar situation exists when a labeled break is used in nested loops with one of the enclosing outer loops being labeled. Control will break out of the enclosing loop to which the labeled break refers. It will be left as an exercise for the student to demonstrate this behavior to his or her satisfaction.

4.9.2 Labeled continue statements

Now consider use of the labeled continue statement. A **continue** statement can only be used in a loop; it cannot be used in a switch. The behavior of a labeled continue statement can be described as follows: "Terminate the current iteration and continue with the next iteration of the loop to which the label refers."

Again, it will be left as an exercise for the student to demonstrate this behavior to his or her satisfaction.

4.10 The return statement

Use of the return statement

Java supports the use of the **return** statement to terminate a method and *(optionally)* return a value to the calling method.

The return type

The type of value returned must match the type of the declared return value for the method.

The void return type

If the return value is declared as **void** , you can use the syntax shown in Figure 8 (p. 15) to terminate the method. *(You can also simply allow the method to run out of statements to execute.)*

An empty return statement.

```
return;
```

Figure 8: An empty return statement.

Returning a value

If the method returns a value, follow the word return with an expression *(or constant)* that evaluates to the value being returned as shown in Figure 9 (p. 15) .

Returning a value from a method.

```
return x+y;
```

Figure 9: Returning a value from a method.

Return by value only

You are allowed to return only by *value* . In the case of primitive types, this returns a copy of the returned item. In the case of objects, returning by value returns a copy of the object's reference.

What you can do with a copy the object's reference

Having a copy of the reference is just as good as having the original reference. A copy of the reference gives you access to the object.

When Java objects are destroyed

All objects in Java are stored in dynamic memory and that memory is not overwritten until all references to that memory cease to exist.

Java uses a garbage collector running on a background thread to reclaim memory from objects that have become *eligible for garbage collection* .

An object becomes eligible for garbage collection when there are no longer any variables, array elements, or similar storage locations containing a reference to the object. In other words, it becomes eligible when

there is no way for the program code to find a reference to the object.

4.11 Exception handling

Exception handling is a process that modifies the flow of control of a program, so it merits being mentioned in this module. However, it is a fairly complex topic, which will be discussed in detail in future modules.

Suffice it at this point to say that whenever an exception is detected, control is transferred to exception handler code if such code has been provided. Otherwise, the program will terminate. Thus, the exception handling system merits being mentioned in discussions regarding flow of control.

5 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [2]
- Java OOP: Objects and Encapsulation [3]

6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0230: Java OOP: Flow of Control
- File: Jb0230.htm
- Originally published: 1997
- Published at cnx.org: November 24, 2012

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[2] http://cnx.org/content/m44148
[3] http://cnx.org/content/m44153