Flow Controls in Java

Java uses all of C's execution control statements, so if you've programmed with C or C++ then most of what you see will be familiar. Most procedural programming languages have some kind of control statements, and there is often overlap among languages. In Java, the keywords include if-else, while, do-while, for, and a selection statement called switch. Java does not, however, support the much-maligned goto (which can still be the most expedient way to solve certain types of problems). You can still do a goto-like jump, but it is much more constrained than a typical goto.

There are three kinds of control statements.

1. Selection
2. Iteration
3. Jump

Selection (If, If-Else-If, Switch)

Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or state of a variable.

If-else

It can be used to route program execution through two different paths.

```
if(Boolean-expression)
  statement;
```

```
if(Boolean-expression){
  statement;
}
```

Or

```
if(Boolean-expression)
  statement1;
else
  statement2;
```

```
if(Boolean-expression) {
  statement1;
} else {
  statement2;
}
```

The if-else statement is probably the most basic way to control program flow. The else is optional. The conditional must produce a Boolean result. The statement means either a simple statement terminated by a semicolon or a compound statement, which is a group of simple statements enclosed in braces. Anytime the word “statement” is used, it always implies that the statement can be simple or compound.

```
int age = 20;
if(age>18){
       System.out.println(“You are a adult.”);
} else {
       System.out.println(“You are a child.”);
}
```

Out put

```
You are a adult.
```

If-else-if

```
if(Boolean-expression1)
  statement1;
else if(Boolean-expression2)
  statement2;
else
  statement3;
```

```
if(Boolean-expression1) {
  statement1;
} else if(Boolean-expression2) {
  statement2;
} else {
  statement3;
}
```

Switch

The switch is sometimes classified as a selection statement. The switch statement selects from among pieces of code based on the value of an integral expression. Its form is:

```
switch(integral-selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; break;
  case integral-value3 : statement; break;
  case integral-value4 : statement; break;
  case integral-value5 : statement; break;
          // ...
  default: statement;
}
```

Integral-selector is an expression that produces an integral value. The switch compares the result of integral-selector to each integral-value. If it finds a match, the corresponding statement (simple or compound) executes. If no match occurs, the default statement executes.

You will notice in the above definition that each case ends with a break, which causes execution to jump to the end of the switch body. This is the conventional way to build a switch statement, but the break is optional. If it is missing, the code for the following case statements execute until a break is encountered. Although you don't usually want this kind of behavior, it can be useful to an experienced programmer. Note the last statement, following the default, doesn't have a break because the execution just falls through to where the break would have taken it anyway. You could put a break at the end of the default statement with no harm if you considered it important for style's sake.

The switch statement is a clean way to implement multi-way selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to an integral value such as int or char. If you want to use, for example, a string or a floating-point number as a selector, it won't work in a switch statement. For non-integral types, you must use a series of if statements.

```
// Demonstrates the switch statement.

public class SwichEx {
  public static void main(String[] args) {
      int c = 2;
      switch(c) {
      case '0':
            System.out.println("number is zero.");
            break;
      case '1':
            System.out.println("number is one.");
            break;
      case '2':
            System.out.println("number is two.");
            break;
      case '3':
            System.out.println("number is three.");
            break;
      default:
            System.out.println("you want to select within 0-3.");
      }
  }
}
```

Out put

```
number is two.
```

If you omit `break;` *Out put*

```
number is two.
number is three.
you want to select within 0-3.
```

Iteration (while, do-while, for)

while, do-while and for control looping and are sometimes classified as iteration statements. A statement repeats until the controlling Boolean-expression evaluates to false.

While

The form for a while loop is

```
while(Boolean-expression)
  statement
```

```
while(Boolean-expression) {
  statement
}
```

The Boolean-expression is evaluated once at the beginning of the loop and again before each further iteration of the statement. If condition is false never go through the body of loop even once.

Here's a simple example that generates random numbers until a particular condition is met:

```
// Demonstrates the while loop.

public class WhileTest {
  public static void main(String[] args) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
    }
  }
}
```

Out put

```
Each time you run this program you'll get a different-sized list of numbers
```

This uses the static method random( ) in the Math library, which generates a double value between 0 and 1. (It includes 0, but not 1.) The conditional expression for the while says “keep doing this loop until the number is 0.99 or greater.” Each time you run this program you’ll get a different-sized list of numbers.

Do-While

The form for do-while is

```
do
  statement;
while(Boolean-expression);
```

```
do {
  statement;
} while(Boolean-expression);
```

The sole difference between while and do-while is that the statement of the do-while always executes at least once, even if the expression evaluates to false the first time. In a while, if the conditional is false the first time the statement never executes. In practice, do-while is less common than while.

```
// Demonstrates the do while loop.

public class DoWhileTest {
  public static void main(String[] args) {
    double r = 0;
    do{
      r = Math.random();
      System.out.println(r);
    } while(r < 0.99d);
  }
}
```

Out put

```
Each time you run this program you’ll get a different-sized list of numbers .But
first value is printed unconditionally.
```

For

A for loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of “stepping.” The form of the for loop is:

```
for(initialization; Boolean-expression; step)
  statement
```

```
for(initialization; Boolean-expression; step) {
  statement

}
```

Any of the expressions initialization, Boolean-expression or step can be empty. The expression is tested before each iteration, and as soon as it evaluates to false execution will continue at the line following for statement. At the end of each loop, the step executes.
for loops are usually used for “counting” tasks:

```
// Demonstrates "for" loop by listing
// all the ASCII characters.

public class ForEx {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
      System.out.println("value: " + (int)c + " character: " + c);
  }
}
```

Out put

```
listing all the ASCII characters up to 128.
```

Note that the variable c is defined at the point where it is used, inside the control expression of the for loop, rather than at the beginning of the block denoted by the open curly brace. The scope of c is the expression controlled by the for.

Traditional procedural languages like C require that all variables be defined at the beginning of a block so when the compiler creates a block it can allocate space for those variables. In Java and C++ you can spread your variable declarations throughout the block, defining them at the point that you need them. This allows a more natural coding style and makes code easier to understand.

You can define multiple variables within a for statement, but they must be of the same type:

```
for(int i = 0, j = 1; i < 10 && j != 11; i++, j++) {
  /* body of for loop */;
}
```

The int definition in the for statement covers both i and j. The ability to define variables in the control expression is limited to the for loop. You cannot use this approach with any of the other selection or iteration statements.

Jump (break, continue, return)

Break and Continue

Inside the body of any of the iteration statements you can also control the flow of the loop by using break and continue. Break quits the loop without executing the rest of the statements in the loop. continue stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

This program shows examples of break and continue within for and while loops:

```
// Demonstrates break and continue keywords.

public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.println(i);
    }
    int i = 0;
    // An "infinite loop":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.out.println(i);
    }
  }
}
```

In the for loop the value of i never gets to 100 because the break statement breaks out of the loop when i is 74. Normally, you'd use a break like this only if you didn't know when the terminating condition was going to occur. The continue statement causes execution to go back to the top of the iteration loop (thus incrementing i) whenever i is not evenly divisible by 9. When it is, the value is printed.

The second portion shows an “infinite loop” that would, in theory, continue forever. However, inside the loop there is a break statement that will break out of the loop. In addition, you’ll see that the continue moves back to the top of the loop without completing the remainder.

(Thus printing happens in the second loop only when the value of i is divisible by 10.)

Out put

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

The value 0 is printed because 0 % 9 produces 0.
A second form of the infinite loop is for(;;). The compiler treats both while(true) and for(;;) in the same way so whichever one you use is a matter of programming taste.

Return

The return keyword has two purposes: it specifies what value a method will return (if it doesn't have a void return value) and it causes that value to be returned immediately. The test( ) method above can be rewritten to take advantage of this:

There's no need for else because the method will not continue after executing a return.

```
public class ReturnEx {
  static int test(int testval, int target) {
    int result = 0;
    if(testval > target)
      return +1;
    else if(testval < target)
      return -1;
    else
      return 0; // Match
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
  }
}
```

Out put

```
+1
-1
0
```