1 Comparison Operators

In addition to the arithmetic and bitwise operators, Java has *comparison* operators, which can be used to compare two primitive data types.

Relational Operator	Meaning
$>$	Greater Than
$<$	Less Than
$>=$	Greater Than or Equal To
$<=$	Less Than or Equal To
$==$	Equal To
$!=$	Not Equal To

Each of these operators produces a value that is either true or false, which can be used in making decisions.

2 The if Statement

The if statement in Java has the following general forms:

```
if (<condition>)
    <if-statement>;
else
    <else-statement>;
```

In this form, if <condition> is true, the <if-statement> will be executed. Otherwise, the <else-statement> will be executed. However, more often than not, the <if-statement> needs to be more than a single statement. In that case, you should use a *statement block*:

```
if (<condition>) {
    <if-statements>;
    ....
    ....
} else {
    <else-statements>;
    ....
    ....
}
```

In general, it is a good idea to use braces even for single statements, since when nested if-else blocks are used, it is easier to follow which else statement corresponds to which if statement when braces are present.

3 Boolean Operators

Java supports the following boolean operators for operating on two boolean values:

<table>
<tr><th>Boolean Operator</th><th>Long Name</th></tr>
<tr><td>&</td><td>Logical AND</td></tr>
<tr><td>&&</td><td>Conditional AND</td></tr>
<tr><td>|</td><td>Logical OR</td></tr>
<tr><td>||</td><td>Conditional OR</td></tr>
<tr><td>!</td><td>Logical NOT</td></tr>
</table>

Note that there are *two* kinds of OR and *two* kinds of AND operators. Their final results are the same, but the side effects are not.

Let us consider the following two AND operations:

```
<expression1> & <expression2>
<expression1> && <expression2>
```

In the first case (logical AND), both expressions will be evaluated, and the result will be true if both expressions evaluate to true. In the second case (conditional AND), expression1 will be evaluated first. If it is false, expression2 will *not* be evaluated at all. (Note that the overall expression is guaranteed to be false if we know that expression1 is false.) If expression2 has any side effects (it may, in general, contain method calls and the such) they will not be evaluted in the second case.

The case for the OR operators is similar:

```
<expression1> | <expression2>
<expression1> || <expression2>
```

In the first case, both expressions will be evaluated. In the second case, expression2 will be evaluated only if expression1 evaluates to be false.

4 The Conditional Operator

The conditional operator is the only operator which takes *three* operands. Its general form is:

```
<logical expression> ? <expression1> : <expression2>
```

The <logical expression> is evaluated first. If it is true, the overall value of the expression is equal to <expression1>. Otherwise, it is equal to <expression2>.

To better understand how it works, note that the following two fragments of code are equavalent:

```
c = a > b ? a : b;

if (a > b) {
    c = a;
} else {
    c = b;
}
```

Note that, while you can always write the equivalent code using if-else blocks, the conditional operator is an operator, and not a statement. Thus, it can be used inside other expressions, while the if-else block can not.

5 The switch Statement

The switch statement allows the program to make a multiple-choice decision depending on the value of a given expression. The expression must evaluate to a char, byte, short or int (but not long, boolean or any floating point value). Suppose, for example, you want to print the words “one”, “two”, “three” and “four” depending on the value of the variable x (which is an int). The below code fragment will achieve this goal:

```
switch(x) {
    case 1:
        System.out.println("one");
        break;
    case 2:
        System.out.println("two");
        break;
    case 3:
        System.out.println("three");
        break;
    case 4:
        System.out.println("four");
        break;
    default:
        System.out.println("Unhandled case.");
        break;
}
```

The execution continues at the case statement that matches the value of the switch variable, and continue down until it sees a break statement, or reaches the end of the switch statement. If you miss the break statement between cases, execution will simply fall through - it will not stop at the next case label. Note that you can also supply a default case where execution will jump to if it matches no other case statement. Also note that the last break statement is not required.

6 Loops

Java supports three sorts of loop statements.

6.1 The `for` Loop

The general form of the for loop is as follows:

```
for (<initialization statement>; <condition>; <increment expression>) {
    <statements>
}
```

The for loop has three parts, separated by commas. The initialization statement is usually used to initialize a loop variable. The condition is evaluated *before* the execution of the body of the for loop, and the body of the for loop is executed if only if the condition evaluates to true. After every execution of the for loop body, the increment expression is executed.

In Java, a lot of times the loop variable is both declared and initialized in the initialization statement. A typical use of the for loop is below:

```
for (int i=1; i < 100; i++) {
    System.out.println(i);
}
```

This will print all integers from 1 to 99.

6.2 The `while` Loop

The general form of the while loop is as follows:

```
while (<condition>) {
    <statements>
}
```

The condition is evaluated, and if it is true, the while loop body is executed. This continues until the expression evaluates to false.

6.3 The `do/while` Loop

The general form of the do/while loop is as follows:

```
do {
    <statements>
} while (<condition>);
```

This is very similar to the `while` loop. But, the condition is evaluated at the end of the loop, i.e., after the statements are executed. This means that the `do/while` loop body is executed at least once, even the condition is always false.

6.4 The continue Statement

The `continue` statement is used when one needs to stop the execution of a loop body and continue with the next iteration of the loop. Here is an example:

```
for(int i=1; i<100; i++) {
    if (i % 2 == 0)
        continue;

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

The `continue` statement will be executed if i is an even number and the rest of the loop will be skipped. Therefore, only odd numbers will be printed.

There is also a *labeled* form of the continue statement. First, you need to *label* the loop statement that you want to continue with the next iteration of. The label is just an identifier followed by a colon which is placed before the loop statement. Then, you use continue followed with the label name to continue at the next iteration of the labeled loop.

6.5 Using the `break` Statement in Loops

The `break` statement, which is used in the `switch` statement, can also be used in loops. When a `break` statement is executed inside a loop, the execution of the loop will be stopped, and the program will continue with the next statement following the loop body.