Decisions in Java – IF Statements

Boolean Values & Variables

In order to make decisions, Java uses the concept of `true` and `false`, which are *boolean values*. Just as is the case with other primitive data types, we can create boolean variables to hold these values.

```
boolean readyToProgram = true;
```

Boolean Expressions

A *boolean expression* is similar to a mathematical expression, except that the result is `true` or `false`, rather than a numeric value. To create a boolean expression, we use the relational operators to compare the values of various data types, such as integers, floats, characters, or strings, using a *relational expression*.

Relational Operator	Meaning	Example	Result
==	is equal to	5 == 5	TRUE
!=	is not equal to	5 != 5	FALSE
<	is less than	3 < 7	TRUE
<=	is less than or equal to	4 <= 4	TRUE
>	is greater than	3 > 7	FALSE
>=	is greater than or equal to	7 >= 3	TRUE

The following rules apply to the use of relational operators with different data types:

1. Values of any of the primitive numeric data types (e.g., int, float, and all their variations) can be used with any of the relational operators.
2. Boolean values can only be tested as “equal to” or “not equal to”.
3. Values of type char are ordered according to the Unicode encoding system. A character the occurs earlier in the system is “less than” a character that occurs later in the system.
   a) For alphabetic characters, this means that 'a' is less than 'z', and 'A' is less than 'Z', as expected.
   b) In the Unicode system, all uppercase letters occur earlier than all lowercase letters. Thus we get the relational ordering of:
   `'A' < 'B' < 'C' < ... < 'Z' < 'a' < 'b' < 'c' < ... <'z'`
   c) Representing numbers as characters, such as when you type on a keyboard, keeps the same ordering, so that `'0' < '1' < '2' < ... < '9'`.

Comparing Strings

The *relational operators* used with the primitive data types should not be used to compare strings. Recall that a `String` variable does not actually contain the string, but only the location (address) of the string in memory. Thus any comparison between strings using relational operators would actually be comparing two addresses, rather than the `String` values.

Java provides a number of *methods* for comparing strings, but for now we will introduce only the `equals` method. This is a *String method*, which means it can be called from any *String variable*.

For example, given the declaration

```
String s = "Same";
```

then the `equals` method would yield the following results:

a) `s.equals("Same")` will return true

b) `s.equals("same")` will return false because 'S' is not equal to 's'

c) `s.equals("Same  ")` will return false because of the spaces at the end of the word

Boolean Operators

It is possible to combine two or more *boolean values, variables, or expressions* into a more complicated boolean expression using the *boolean operators*. Unlike the *relational operators*, the boolean operators can only work on boolean values. You can use a relational operator to compare any data type, which forms a boolean expression. Multiple boolean expressions can be combined using boolean operators.

The boolean operators are summarized in the following table.

boolean value		**not p**	**p AND q**	**p OR q**
p	**q**	**!p**	**p && q**	**p || q**
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	TRUE

1. `!p` (not p) has the true/false value opposite to p.
2. `p && q` (p AND q) is true if and only if both p and q are both true.
3. `p || q` (p OR q) is true if p is true, q is true, or both p and q are true.

One Choice – IF-THEN Statements

```
if (<boolean expression>)
{
  <statements>
}
```

The simplest decision is to choose between doing something, or doing nothing. In this general code outline, the `<boolean expression>` may be quite simple or very sophisticated. Similarly, the `<statements>` could be a single line of Java code, or a very complicated block of code. Nonetheless, there is a single decision here – if the boolean expression evaluates to true, the statements will be executed. If the expression is false, no code from this block is executed.

Two Choices – IF-THEN-ELSE Statements

Rather than doing *nothing*, it is far more common to use our decision to choose between two possible options – one for `true`, one for `false`. The additional code, for the false condition, is placed within the `else` block of the statement. This is the most commonly occurring decision statement.

```
class IfDemo
{
  public static void main (String[] args)
  {
    System.out.print("Please give one integer: ");
    int first = In.getInt();
    System.out.print("and a second: ");
    int second = In.getInt();

    if (first == second)
    {
      System.out.println("The values are equal");
    }
    else
    {
      System.out.println("The values are not equal");
    }
  }
}
```

Many Choices – Nested IF Statements

It is possible to make more than two choices with multiple if statements:

```
if (x < 0)
{
  System.out.println("value is negative);
}
if (x > 0)
{
  System.out.println("value is positive);
}
if (x == 0)
{
  System.out.println("value is zero);
}
```

This code involves three tests, which may not seem significant now, but inefficient testing of conditions is one of the major sources of slow and inefficient programs. The following code improves the efficiency by reducing the code to a maximum of two comparisons using *nesting*.

```
if (x < 0)
{
  System.out.println("value is negative);
}
else
{
  if (x > 0)
  {
    System.out.println("value is positive);
  }
  else
  {
    System.out.println("value is zero);
  }
}
```

An alternate format is also available which has a more linear appearance than the nested code.

```
if (x < 0)
{
  System.out.println("value is negative);
}
else if (x > 0)
{
  System.out.println("value is positive);
}
else // (x == 0) is the only remaining option
{
  System.out.println("value is zero);
}
```