Part 3

JavaScript Comparison Operators

As well as needing to assign values to variables, we sometime need to compare variables or literals.

We do this using *Comparison Operators*.

Comparison Operators compare two values and produce an output which is either true or false.

For example, suppose we have two variables which are both numeric (i.e., they both hold numbers):

```
examPasses
```

and

```
totalStudents
```

If we compare them, there are two possible outcomes:

- They have the same value
- They do not have the same value

Therefore, we can make statements like these:

- They are the same
- They are the different
- The first is larger than the second
- The first is smaller than the second

... and then perform a comparison to determine whether the statement is true or false.

The basic comparison operator is:

```
==
```

(i.e., two equals signs, one after the other with no space in between).

It means 'is equal to'. Compare this with the *assignment operator*, =, which means 'becomes equal to'. The assignment operator *makes* two things equal to one another, the comparison operator tests to see if they are *already* equal to one another.

Here's an example showing how the comparison operator might be used:

```
examPasses == totalStudents
```

If `examPasses` and `totalStudents` have <u>the same</u> value, the comparison would return `true` as a result.

If `examPasses` and `totalStudents` have <u>different</u> values, the comparison would return `false` as a result.

Another comparison operator is:

!=

(i.e., an exclamation mark followed by an equals sign, with no space in between).

It means 'is NOT equal to'.

For example:

```
examPasses != totalStudents
```

If `examPasses` and `totalStudents` have <u>the same</u> value, the comparison would return `false` as a result.

If `examPasses` and `totalStudents` have <u>different</u> values, the comparison would return `true` as a result.

Two other commonly-used comparison operators are:

<

and

>

<	means 'less than'
>	means 'greater than'

For example:

```
examPasses < totalStudents
```

> If `examPasses` is <u>less than</u> `totalStudents`, the comparison would return `true` as a result.
>
> If `examPasses` is <u>more than</u> `totalStudents`, the comparison would return `false` as a result.

Another example:

```
examPasses > totalStudents
```

> If `examPasses` is <u>more than</u> `totalStudents`, the comparison would return `true` as a result.
>
> If `examPasses` is <u>less than</u> `totalStudents`, the comparison would return `false` as a result.

As with some of the other 0perators we have encountered, comparison operators can be combined in various ways.

`<=`

means

'less than or equal to'.

For example:

```
examPasses <= totalStudents
```

> If `examPasses` is <u>less than or equal to</u> `totalStudents`, the comparison would return `true` as a result.
>
> If `examPasses` is <u>more than</u> `totalStudents`, the comparison would return `false` as a result.

Also:

`>=`

means

'greater than or equal to'.

For example:

```
examPasses >= totalStudents
```

If `examPasses` is <u>more than or equal to</u> `totalStudents>`, the comparison would return `true` as a result.

If `examPasses` is <u>less than</u> `totalStudents`, the comparison would return `false` as a result.

To summarise, JavaScript understands the following comparison operators:

==	'is equal to'
!=	'is NOT equal to'
<	'is less than'
>	'is greater than'
<=	'is less than or equal to'
>=	'is greater than or equal to'

If-Else Statements in JavaScript

Much of the power of programming languages comes from their ability to respond in different ways depending upon the data they are given.

Thus all programming languages include statements which make 'decisions' based upon data.

One form of decision-making statement is the `If...Else` statement.

It allows us to make decisions such as:

If I have more than £15 left in my account, I'll go to the cinema.

Otherwise I'll stay at home and watch television.

This might be expressed in logical terms as:

If (money > 15) go_to_cinema

Else watch_television

The If-Else statement in JavaScript has the following syntax:

```
if (condition)
{
```

```
                statement;
                statement
        }
        else
        {
                statement;
                statement
        };
```

The condition is the information on which we are basing the decision. In the example above, the condition would be whether we have more than £15. If the condition is true, the browser will carry out the statements within the `if...` section; if the condition is false it will carry out the statements within the `else...` section.

The `if...` part of the statement can be used on its own if required. For example:

```
        if (condition)
        {
                statement;
                statement
        };
```

Note the positioning of the semi-colons.

If you are using both the `if...` and the `else...` parts of the statement, it is important NOT to put a semi-colon at the end of the `if...` part. If you do, the `else...` part of the statement will never be used.

A semi-colon is normally placed at the very end of the `if...else...` statement, although this is not needed if it is the last or only statement in a function.

A practical If-Else statement in JavaScript might look like this:

```
        if (score > 5)
        {
                alert("Congratulations!")
        }
        else
        {
                alert("Shame - better luck next time")
        };
```

The `for` Loop

A `for` loop allows you to carry out a particular operation a fixed number of times.

The `for` loop is controlled by setting three values:

- an initial value
- a final value
- an increment

The format of a `for` loop looks like this:

```
for (initial_value; final_value; increment)
{
    statement(s);
}
```

A practical `for` loop might look like this:

```
for (x = 0; x <= 100; x++)
{
    statement(s);
}
```

Note that:

* The loop condition is tested using a variable, `x`, which is initially set to the start value (`0`) and then incremented until it reaches the final value (`100`).

* The variable may be either incremented or decremented.

* The central part of the condition, the part specifying the final value, must remain true throughout the required range. In other words, you could not use `x = 100` in the `for` loop above because then the condition would only be true when x was either 0 or 100, and not for all the values in between. Instead you should use `x <= 100` so that the condition remains true for all the values between 0 and 100.

Here are some practical examples of `for` loops.

The first example is a simple loop in which a value is incremented from 0 to 5, and reported to the screen each time it changes using an alert box. The code for this example is:

```
for (x = 0; x <= 5; x++)
{
    alert('x = ' + x);
}
```

Count Up Click here to see this example working.

The second example is the same except that the value is decremented from 5 to 0 rather than incremented from 0 to 5. The code for this example is:

```
for (x = 5; x >= 0; x--)
{
    alert('x = ' + x);
}
```

Count Down Click here to see this example working.

The start and finish values for the loop must be known before the loop starts.

However, they need not be written into the program; they can, if necessary, be obtained when the program is run.

For example:

```
var initialValue = prompt("Please enter initial value", "");
var finalValue = prompt("Please enter final value", "");

for (x = initialValue; x <= finalValue; x++)
{
    statement(s);
}
```

In this example, the user is prompted to type-in two numbers which are then assigned to the variables `initialValue` and `finalValue`. The loop then increments from `initialValue` to `finalValue` in exactly the same way as if these values had been written directly into the program.

The `While` Loop

Like the `for` loop, the `while` loop allows you to carry out a particular operation a number of times.

The format of a `while` loop is as follows:

```
while (condition)
{
    statement(s);
}
```

A practical `while` loop might look like this:

```
    var x = 500000;

    alert("Starting countdown...");

    while (x > 0)
    {
        x--;
    };
    alert("Finished!");
```

In this example, `x` is initially set to a high value (`500,000`). It is then reduced by one each time through the loop using the decrement operator (`x--`). So long as `x` is greater than zero the loop will continue to operate, but as soon as `x` reaches zero the loop condition (`x > 0`) will cease to be true and the loop will end.

The effect of this piece of code is to create a delay which will last for as long as it takes the computer to count down from 500,000 to 0. Before the loop begins, an 'alert' dialog-box is displayed with the message "Starting Countdown...". When the user clicks the 'OK' button on the dialog-box the loop will begin, and as soon as it finishes another dialog-box will be displayed saying "Finished!". The period between the first dialog box disappearing and the second one appearing is the time it takes the computer to count down from 500,000 to 0.

To see this example working, click here. Delay

The principal difference between `for` loops and `while` loops is:

> with a `while` loop, the number of times the loop is to be executed need not be known in advance.

`while` loops are normally used where an operation must be carried out repeatedly until a particular situation arises.

For example:

```
    var passwordNotVerified = true;

    while (passwordNotVerified == true)
    {
        var input = prompt("Please enter your password", "");
        if (input == password)
        {
            passwordNotVerified = false;
        }
        else
        {
            alert("Invalid password - try again")
        }
    }
```

In this example, the variable `passwordNotVerified` is initially set to the Boolean value `true`. The user is then prompted to enter a password, and this password is compared with the correct password stored in the variable called `password`. If the password entered by the user matches the stored password, the variable `passwordNotVerified` is set to `false` and the `while` loop ends. If the password entered by the user does not match the stored password, the variable `passwordNotVerified` remains set to `true` and a warning message is displayed, after which the loop repeats.

To try this piece of code, click here. Check Password

PS The password is `CS7000` - and don't forget that the 'CS' must be capitalised.

Testing Boolean Variables

In the `while` loop example above we used the line:

```
  var passwordNotVerified = true;
```

and then tested this variable in a conditional statement as follows:

```
  while (passwordNotVerified == true)
```

We could also have written the conditional statement like this:

```
  while(passwordNotVerified)
```

In other words, if we don't specify `true` or `false` in a conditional statement, the JavaScript interpreter will assume we mean `true` and test the variable accordingly.

This allows us to make our code a little shorter and, more importantly, to make it easier for others to understand. The line:

```
  while(passwordNotVerified)
```

is much closer to the way in which such a condition might be expressed in English than:

```
  while(passwordNotVerified == true)
```

Logical Operators

We have met a number of operators that can be used when testing conditions, e.g., == , < , > , <= , >= .

Two more operators that are particularly useful with `while` loops are:

`&&`	Logical AND
`||`	Logical OR

These operators are used to combine the results of other conditional tests.

For example:

```
if (x > 0 && x < 100)
```

means...

```
if x is greater than 0 and less than 100...
```

Placing the `&&` between the two conditions means that the `if` statement will only be carried out if BOTH conditions are true. If only one of the conditions is true (e.g., x is greater than 0 but also greater than 100) the condition will return false and the `if` statement won't be carried out.

Similarly:

```
if (x == 0 || x == 1)
```

means...

```
if x is 0 or x is 1...
```

Placing the `||` between the two conditions means that the `if` statement will be executed if EITHER of the conditions are true.

The ability to combine conditions in this way can be very useful when setting the conditions for `while` loops.

For example:

```
var amount = prompt ("Please enter a number between 1 and 9", "");

while (amount < 1 || amount > 9)
{
    alert("Number must be between 1 and 9");
    amount = prompt ("Please enter a number between 1 and 9", "");
}
```

In this example, the variable `amount` is initially set to the value typed-in by the user in response to the 'prompt' dialog-box. If the amount entered by the user is between 1 and 9, the loop condition becomes `false` and the loop ends. If the amount entered by the user is less than 1 or greater than 9, the loop condition remains `true` and a warning is displayed, after which the user is prompted to enter another value.

To try this piece of code, click here. Check Number

Note that if you enter a correct value immediately, the `while` loop never executes. The condition is false the first time it is tested, so the loop never begins.