Part 6

The `String` Object

JavaScript includes a `String` object that allow us to manipulate strings in a variety of ways, for example, searching a string to see if it contains certain patterns of letters, extracting part of it to form new string, and much more.

A `String` object is created in the following way:

```
var myString = new String("Hello World");
```

However, most browsers regard *any* string as an instance of the `String` object. Therefore you can declare a string in the normal way, e.g.:

```
var myString = "Hello World";
```

...and in so doing you will automatically create a `String` object, complete with its associated methods, properties, etc..

`String` *properties* include:

`length` Returns the length of a string. For example, here is a text box:

It is called `textBox1` and is part of a form called `form1`.

If someone types into the box, you could find out how many characters they typed using the following code:

```
var stringLength =
document.form1.textBox1.value.length
```

Type some text into the box, then click here to see this code in operation. Display String Length

`String` *methods* include:

`charAt()` Returns the character at a specified position in a string. The syntax is:

```
charAt(index)
```

where `index` is a number representing the position of a character in the string. For example, here is a text box:

It is called `textBox2` and is part of `form1`.

You could find out what the third character in the text-box is using the following code:

```
var thirdCharacter =
document.form1.textBox2.value.charAt(2);
```

(Note that the characters in a string are numbered from zero, not one. Therefore the third character will be character number two.)

Type some text into the box, then click here to see this code in operation. Find Third Chararacter

`indexOf()`

Searches a string to see if it contains a specified character, and if it does, returns the position of that character. If the specified character occurs more than once in the string, it returns the position of the first occurrence of that character. The syntax is:

```
indexOf(character)
```

For example, here is a text box:

It is called `textBox3` and is part of `form1`.

To find out whether the text-box contains the letter 'c', we could use the following code:

```
var positionOfC =
document.form1.textBox3.value.indexOf("c");
```

Type some text into the box, then click here to see this code in operation. Find position of C

Again, bear in mind that the characters in the string are numbered from zero, not one, so the index will be one less than the character's actual position. Also note that if there is no 'c' in the string, the value returned will be -1.

`lastIndexOf()` Searches a string to see if it contains a specified character, and if it does, returns the position of that character. It performs the same function as `indexOf()`, except that if the specified character occurs more than once in the string, it returns the position of the *last* occurrence of that character rather than the first.

`substring()` Returns the portion of a string between two specified positions. The syntax is:

```
substring(start_position, end_position)
```

where `start_position` is the position of the first character in the wanted portion of the string (counting from zero), and `end_position` is the position *after* the last character of the wanted portion of the string.

For example, here is a text box:

It is called `textBox4` and is part of `form1`.

Suppose we wish to extract the third, fourth and fifth characters from a string in the text-box. Counting from zero, these would be the characters in positions 2, 3 and 4. Therefore we would set the first parameter to 2 and the second parameter to 5 (one position after the last character we want). For example:

```
var extract =
document.form1.textBox4.value.substring(2,5);
```

Type some text into the box, then click here to see this code in operation. Find 3-5th Characters

`substr()` Returns a portion of a string, starting from a specified positions and continuing for a specified number of characters. The syntax is:

```
substr(start_position, no_of_characters)
```

where `start_position` is the position of the first character in the wanted portion of the string (counting from zero), and `no_of_characters` is the number of characters to extract, starting at that position.

In other words, it behaves in a very similar way to the `substring()` method, except that second parameter specifies the *number* of characters to extract from the string rather than the position after the last character.

`charCodeAt()` Returns the numerical (ASCII) value of the character at the specified position.

For example, here is a text box:

It is called `textBox5` and is part of `form1`.

To find the ASCII value of a single character typed into this box, we could use the following code:

```
var asciiValue =
document.form1.textBox5.value.charCodeAt(0);
```

Type a character into the box, then click here to see this code in operation. Find ASCII Value

`fromCharCode()` Returns the characters represented by a sequence of numerical (ASCII) values.

For example:

```
var asciiString =
String.fromCharCode(73,97,110);
```

This code will create a string containing the ASCII characters whose numerical valies are 73, 97 and 110. Click here to see this code in operation. Convert ASCII Values

`toString()` Converts a number into a string. The syntax is:

```
toString(number)
```

For example, consider the following code:

```
var aNumber = 12345;
alert("The length is " + aNumber.length);

var aNumber = aNumber.toString();
alert("The length is " + aNumber.length);
```

The variable `aNumber` is not enclosed in quotes or otherwise declared as a string, so it is a *numeric* variable. Therefore it doesn't have a `length` property, and the first `alert()` dialog in the example will report that the length is 'undefined'.

However, once we have converted `aNumber` into a string using the `toString()` method, it will have a `length` property just like any other string, so the second `alert()` dialog in the example should report the length correctly.

Click here to see this code in operation. Convert Number To String

In addition to the methods shown above, the `String` object also provides methods to add HTML formatting to strings. These methods include:

`italics()` Formats a string with and tags. The syntax is:

```
string_name.italics()
```

For example:

```
var myString = "Hello World";
document.write(myString.italics()):
```

Would have the same effect as:

```
var myString = "Hello World";
document.write("<i>" + myString +
"</i>"):
```

`bold()` Formats the string with and tags. Works in a similar fashion to the `italics()` method (see above).

`sup()` Formats the string with ^{and} (i.e., super-script) tags. Works in a similar fashion to the `italics()` method (see above).

`sub()`	Formats the string with `<sub>` and `</sub>` (i.e., sub-script) tags. Works in a similar fashion to the `italics()` method (see above).

Regular Expressions

A regular expression describes a pattern of characters or character types that can be used when searching and comparing strings.

For example, consider a web-site that allows the user to purchase goods online using a credit-card. The credit-card number might be checked before submission to make sure that it is of the right type, e.g., that it is 16 digits long, or consists of four groups of four digits separated by spaces or dashes.

It would be quite complicated to perform such checks using just the string-handling functions described above. However, using regular expressions we could create a pattern that means 'four digits followed by a space or a dash, followed by four more digits...', etc.. It is then quite simple to compare this pattern with the value entered by the user and see if they match or not.

The special characters that can be used to create regular expressions include:

\d	Represents any numerical character
{x}	Indicates that the preceding item should occur x times consecutively

In addition, any letter or other character can be used explcitly. For example, if you place an 'a' in a regular expression pattern it means that an 'a' is expected as a match at that point.

Using these pattern-matching characters, we could create a simple pattern that checks for a credit-card number in the following way:

```
\d{4}-\d{4}-\d{4}-\d{4}
```

The first '`\d`' means 'any number'. This is followed by '`{4}`', which extends the pattern to mean 'any four consecutive numbers'. After this comes a '−' which simply means that a dash character is expected at this point. Then comes the 'any four consecutive numbers' pattern again, followed by the dash, and so on.

To create such a pattern, we would declare a regular expression object and give it the pattern as a value. This can be done in two ways:

```
var myRegExp = new RegExp("\d{4}-\d{4}-\d{4}-\d{4}");
```

Or:

```
var myRegExp = /\d{4}-\d{4}-\d{4}-\d{4}/;
```

In the first example, the new regular expression object (`RegExp`) is declared explicitly.

In the second example, the pattern is declared in much the same way a string might be declared, except that forward-slashes (`/`) are used at the beginning and end instead of quote-marks. The forward-slashes indicate that this sequence of characters is being declared as a regular expression rather than as a string.

Once the regular expression has been created, we can use the `test()` method to compare it with a string. For example:

```
var myRegExp = /\d{4}-\d{4}-\d{4}-\d{4}/;

var inputString = prompt("Please enter Credit Card
number","");

var result = myRegExp.test(inputString);

alert(result);
```

In this example, the regular expression is assigned to a variable called `myRegExp`. The user is then prompted to enter a string which is assigned to the variable `inputString`. The `test()` method is then used to compare the two strings, with the result being passed to the variable `result` which is displayed in an `alert()` dialog-box. If the string matches the pattern, the result will be true; if not, it will be false.

Compare Strings Click here to see this example working. Try entering various numbers into the box and see the result. You should find that the code only returns true if you enter a number that consists of four groups of four digits separated by dashes.

We can also compare a string with several different patterns using the Logical OR operator. For example:

```
\d{16}|\d{4}-\d{4}-\d{4}-\d{4}
```

This example is similar to the previous one except that the following characters have been added to the start of the string:

```
\d{16}|
```

The first few characters, `\d{16}`, mean 'any sixteen consecutive numbers'. The `|` character indicates a logical OR, meaning that the whole expression will be true if either the part before this symbol OR the part after it are true.

In other words, a string will be regarded as a valid match if it contains 16 consecutive digits OR four groups of four digits separated by dashes.

Compare Strings Click here to see this example working. You should find that the code returns true if you enter a number that consists of sixteen consecutive digits or four groups of four digits separated by dashes.

This is fine if the user enters consecutive numbers or groups of numbers separated by dashes, but what if the user enters groups of numbers separated by spaces?

It is possible test a character in a string to see if it is any one of several specified characters. This can be done in the following way:

[xyz]	Match any of the characters within the brackets, e.g., if the characters within the brackets are x, y and z, the test will return true if the character at that point in the string is either x, y or z.

Using this method, we could modify our previous pattern as follows:

```
\d{16}|\d{4}[ -]\d{4}[ -]\d{4}[ -]\d{4}
```

In this example, each of the dashes has been replaced with a pair of square brackets containing both a space and a dash. This means that either a space or a dash will be accepted at these points in the string. Thus the complete string will regarded as a valid match if it contains 16 consecutive digits OR four groups of four digits separated by spaces or dashes.

Compare Strings Click here to see this example working. You should find that the code returns true if you enter a number that consists of sixteen consecutive digits or four groups of four digits separated either by dashes or spaces.

The examples given above indicate only a few of the possibilities offered by regular expressions.

Some of the most commonly-used pattern-matching characters are shown below. For a more complete list you should consult a good reference book (the 'Pure JavaScript' book recommended for use with this course has quite an extensive list).

\w	Represents any alphanumerical character
\W	Represents any non-alphanumerical character
\d	Represents any numerical character
\D	Represents any non-numerical character
\s	Represents any 'whitespace' character (e.g., carriage-return, tab, etc.)
\S	Represents any non-whitespace character

[..]	Match any one of the characters within the brackets
[^..]	Match any one character other than those within the brackets
[x-y]	Match any character within the range x to y
[^x-y]	Match any one character other than those within the range x to y
{x}	Match the previous item x times
{x,}	Match the previous item at least x times

Using Regular Expressions with Strings

In addition to the methods described above, the `String` object has a number of methods that are specifically designed to work with regular expressions.

`search(regExp)` Searches for any occurrences of the regular expression in the string. If any are found, it returns an integer indicating where within the string the matching sequence begins. If no match is found it returns -1.

For example:

The textbox above is called `textBox6` and it is part of a form called `form2`.

If someone typed a string into the box containing a two-digit number, you could find where the number begins using the following code:

```
var myRegExp = /\d\d/;

var numStart =
document.form2.textBox6.value.search(myRegExp);

alert("The number starts at position " +
numStart);
```

Type some text into the box, then click here to see this code in operation.
Find Number

`replace(regExp, newString)` Searches for any occurrences of the regular expression in the string. If any are found, it replaces them with `newString`.

For example:

The textbox above is called `textBox7` and it is part of a form called `form2`.

If someone typed a string into the box containing a two-digit number, you could replace the number with 00 using the following code:

```
var myRegExp = /\d\d/;

var newString =
document.form2.textBox7.value.replace(myRegExp,
"00");

alert("The modified string is: " + newString);
```

Type some text into the box, then click here to see this code in operation.
Replace Number