Scripting the DOM

Terry Marris September 2011

1 JavaScript Tutorial

A web page has three components:

- content : Hypertext Markup Language, HTML
- presentation: Cascading Style Sheets, CSS
- behaviour: JavaScript acting on the Document Object Model, DOM

With JavaScript we can create dynamic web pages that react to visitor input. With JavaScript we can respond to mouse clicks over links. With JavaScript we can create picture galleries where the visitor can select an image for inspection. With JavaScript we can validate on-screen forms that site visitors complete.

DOM provides an interface that allows a scripting language such as JavaScript to interact with HTML components.

We focus on using JavaScript and the DOM. A working knowledge of HTML, CSS, a programming language such as C or Java, and an image manipulation program such as GIMP is assumed.

Throughout this series of notes I shall be using Internet Explorer 8, IE8.

1.1 Hello World

The first task is to display *Hello, World!*

The first method for doing so uses the JavaScript *alert()* function to put up a message box.

In general we use HTML5.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Hello World</title>
</head>
<body>
<script src="helloworld.js"></script>
</body>
</html>
```

Usually we place the line that imports the JavaScript file just before *</body>* to help the page to load quickly.

```
<script src="helloworld.js"></script>
```

The JavaScript file contains just two lines:

```
/* helloworld.js */
alert("Hello, World!");
```

The line

```
/* helloworld.js */
```

is a comment. Comments like this start with /* and end with */, and can extend over several lines, but they cannot be nested. You could use // to mark the rest of a line as a comment if you like.

The function call

```
alert("Hello, World!");
```

displays a message box showing the text *Hello, World!* The text is enclosed within double quotation marks. But you could use single quotation marks if you prefer. In general, we terminate each statement with a semi-colon.

The second way of displaying *Hello, World!* uses JavaScript and the DOM to create a paragraph on the HTML page.

The revised JavaScript file, *helloworld.js*, is shown below.

```
/* helloworld.js */

window.onload = function()
{
  write("Hello, World!");
}
```

function() introduces a function with no name and no parameters. A call to a function named *write()* is made with the argument value *Hello, World!*. *write()* is described below. *{* and *}* mark the beginning and end of a block of statements. *window.onload* is the HTML event that occurs when a page has finished loading. So, here we are saying: when the page has finished loading *write("Hello, World!")*. The JavaScript *write()* function is shown below.

```
/* write.js */

/* write: creates a new html paragraph, displays the given text */
function write(text)
{
  var testdiv = document.getElementById("testdiv");
  var para = document.createElement("p");

  var txtNode = document.createTextNode(text);

  para.appendChild(txtNode);
  testdiv.appendChild(para);
}
```

The *write()* function

- retrieves the HTML *div* element named *testdiv* (see the HTML file below).
- creates an HTML paragraph
- creates a text node with the given text
- adds the node with the given text to paragraph element described above
- adds the paragraph element to the HTML *div* named *testdiv*

var introduces a variable definition. Notice that variable types are not declared, and that a parameter's type is also not declared.

function and *var* are both JavaScript words.

getElementById(), *createElement()*, *createTextNode()* and *appendChild()* are all DOM methods. Case is significant. *getelementbyid()* is not the same as *getElementById()*. We shall have more to say about these methods later.

The HTML is shown below.

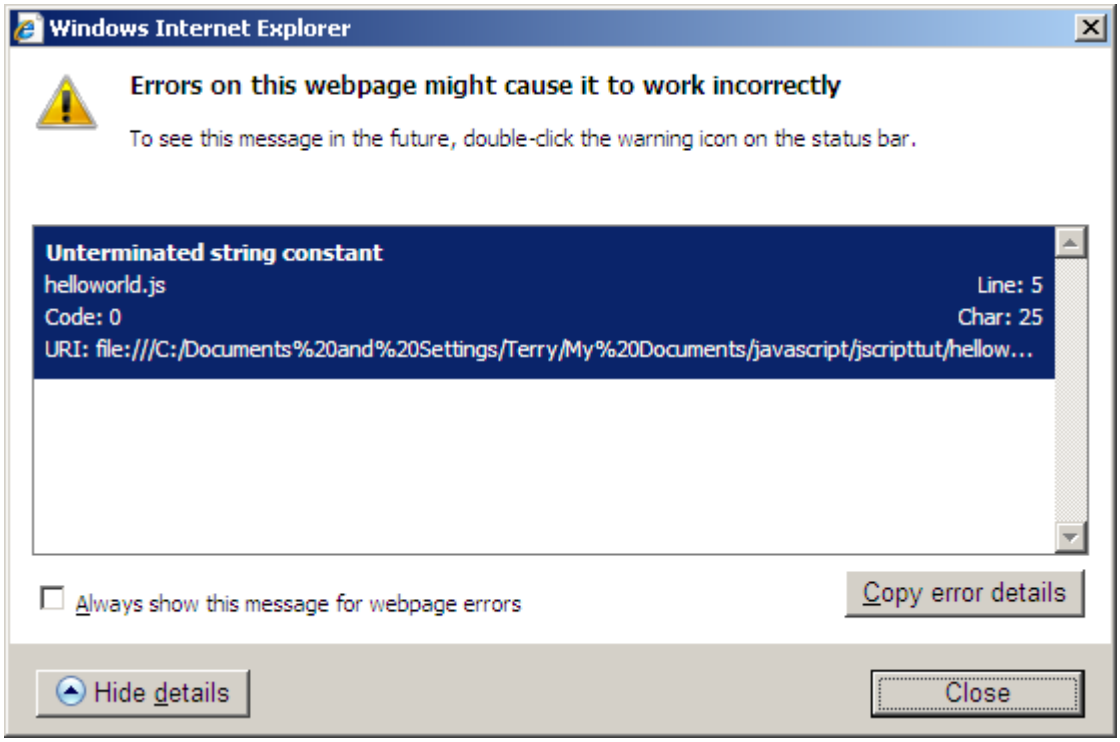
```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Test</title>
</head>

<body>
<div id="testdiv">
</div>

<script src="write.js"></script>
<script src="helloworld.js"></script>

</body>
</html>
```

And here is the result.

1.2 Development Environment

I use Microsoft Notepad for writing JavaScript, HTML and CSS. In classic Windows XP, Notepad can be reached from the Start menu: *Start, Programs, Accessories, Notepad*.

The JavaScript interpreter translates JavaScript text into an executable format, and is part of the web browser, IE8 in my case. The interpreter translates the JavaScript line-by-line until it finds an error such as missing quotation marks, unbalanced braces or a call to a non-existent function. An error is signalled with a small yellow triangular icon on the status bar at the bottom left of the IE8 window. You might need to choose *View, Toolbars* and check *Status Bar* to see the status bar. Double-clicking on this icon will pop up an error message, such as the one shown below.

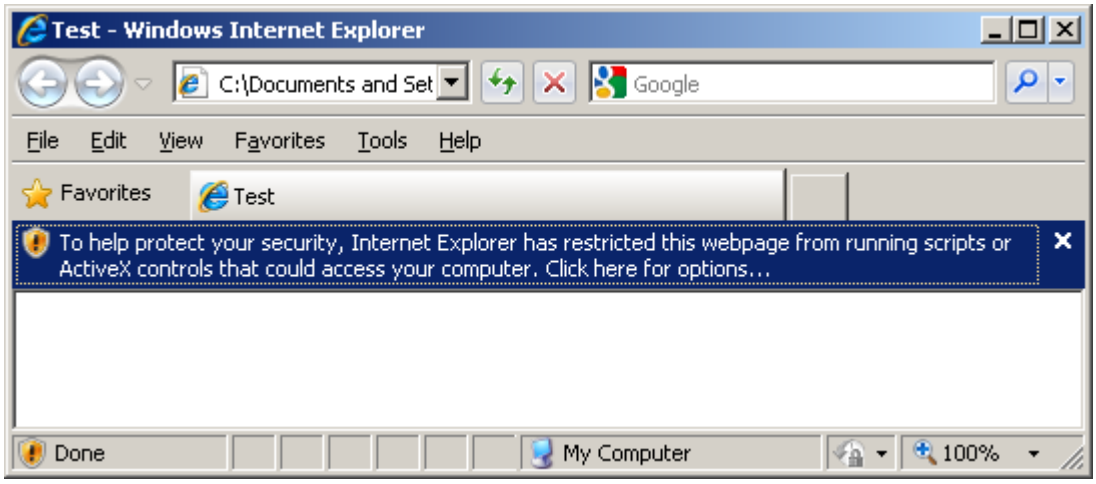


Here you can see that a quotation mark is missing from the end of a string on line 5 of *helloworld.js*.

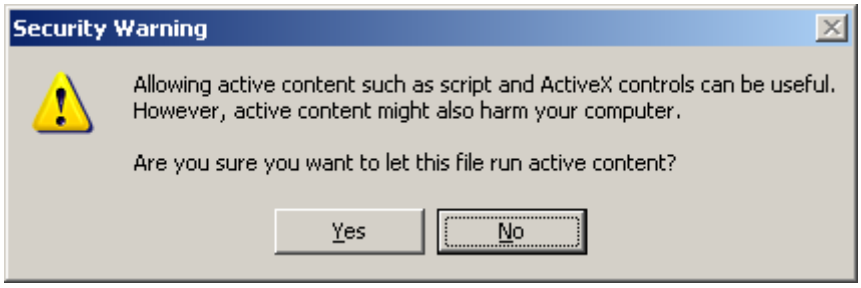
You could use *alert()* to display your own diagnostics to help locate obscure logical errors, where the results you get are not the ones you expected.

```
var v;
...
alert("The value of v is " + v);
```

You will need to *allow blocked content* if IE8 will not run your scripts.



Click on *Click here for options ....* Click on *Allow blocked content ...* Then click on *Yes* in response to *Are you sure you want this file to run active content?*



Exercise 1.1

1. Run the script, shown above, to display *Hello, World!* on your computer system. Determine the effect of leaving out parts such as quotation marks, brackets, semi-colons, and determine what happens when you miss-spell various words such as *testdiv* and *write.js*. Does case (upper and lower case letters) matter?

1.3 Variables and Arithmetic

JavaScript types include strings, numbers and Booleans. A string is a sequence of characters, where characters include letters of the alphabet, digits, and punctuation symbols including space. Numbers include both integers and floating-point numbers. The Boolean values are *true* and *false*.

Types are not explicitly defined in variable declarations. Rather, the type is inferred from the value assigned to the variable.

```
var name = "Jo King";          /* a string */
var ageInYears = 40;           /* an integer */
var weightInKG = 290.5;        /* a float or double */
var isMale = true;             /* a boolean */
```

var introduces a variable declaration.

= is the assignment operator. It copies from right to left.

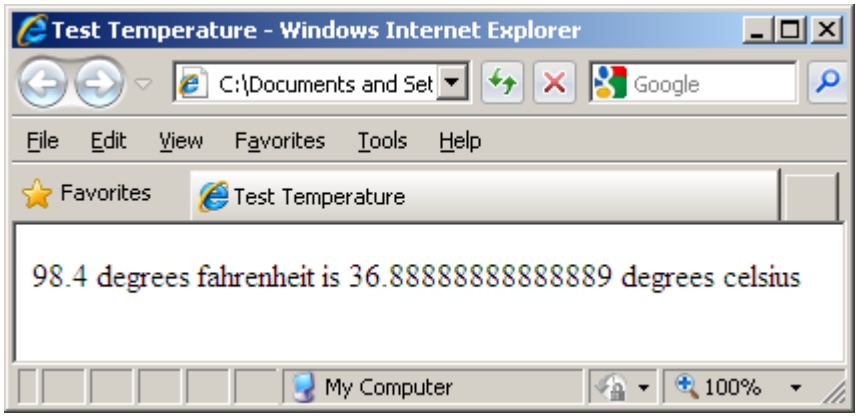
Quotation marks delimit a string literal. You can use either single quotes or double quotes.

true and *false* are both JavaScript words.

The usual rules of arithmetic precedence apply: brackets first, then multiplication and division, then addition and subtraction.

```
var fahr = 98.4
var celsius = (5.0 / 9.0) * (fahr - 32.0);
```

If the value of *celsius* was printed we would expect to see something like 37.0 (near normal human body temperature). In fact we see



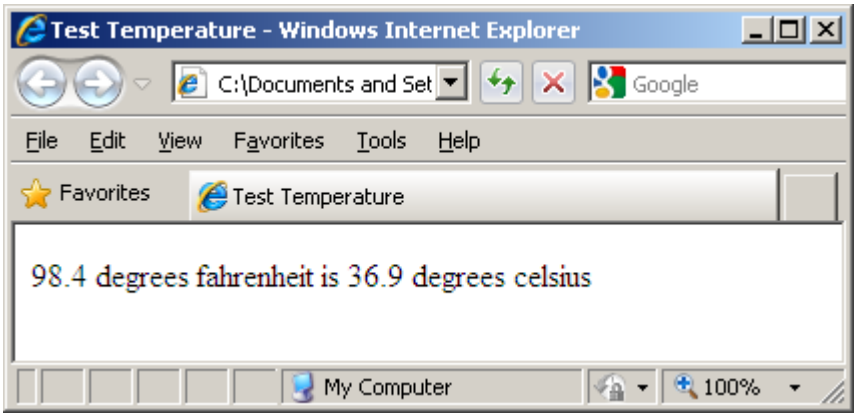
when

```
/* temperature.js */

window.onload = function()
{
  var fahr = 98.4;
  var celsius = (5.0 / 9.0) * (fahr - 32.0);

  write(fahr + " degrees fahrenheit is " +
        celsius + " degrees celsius");
}
```

is executed by

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Test Temperature</title>
</head>

<body>
<div id="testdiv">
</div>

<script src="write.js"></script>
<script src="temperature.js"></script>

</body>
</html>
```

where *write.js* is the JavaScript file introduced in Section 1.1 above. You might agree that the result looks pretty ugly. We need to format the output to, say, one decimal place.



We can do this with the JavaScript *toFixed()* method.

number.toFixed(digits)

number is the object we want to format. *digits* is the number of digits to be shown after the decimal point.

Our improved JavaScript is

```
/* temperature.js */

window.onload = function()
{
  var fahr = 98.4;
  var celsius = (5.0 / 9.0) * (fahr - 32.0);

  write(fahr + " degrees fahrenheit is " +
        celsius.toFixed(1) + " degrees celsius");
}
```

Look at the argument value to *write()*.

```
fahr + " degrees fahrenheit is " +
  celsius.toFixed(1) + " degrees celsius"
```

In this context, + is the concatenation operator; it adds one string onto the end of another, converting number values to strings as it does so.

Look at *celsius.toFixed(1)*. We say that *celsius* is the object receiving the message *toFixed(1)*. We think hey, *celsius*, go and do your *toFixed()* method with the given argument value *1*. If *celsius*, a number value, does not have a *toFixed()* method we have an error.

A method is a function that is bound to an object. A list of inbuilt JavaScript methods can be found at www.tutorialpoint.com.

The increment and decrement operators are ++ and -- respectively.

```
var m = 10;
var n = 10;
```

m++ leaves *m* with the value 11. *n*-- leaves *n* with the value 9.

You often see expressions like

```
var m = 10;
...
m += 2;
```

This adds 2 to *m* leaving *m* with the value 12. *m*+= *2*; is a terse way of writing

```
m = m + 2;
```

You may remember from your junior school days that

7 ÷ 3 = 2 remainder 1

Well, the modulo operator, %, gives you the remainder after division.

7 % 3 = 1

And *7 % 0* is undefined because you cannot divide by zero.

Exercise 1.2

1. You have 50 shares valued at 85.00 US dollars (USD) per share. Write and test a JavaScript script that displays the value of the shares in pounds Sterling (GBP) when 1.00 USD = 0.623 GBP.

1.4 Selections

We have the usual relational operators:

<	less than
<=	less than or equal to
>	more than
>=	more than or equal to

The following expressions are all true: 2 < 3, 2 <= 3, 2 <= 2, 3 > 2, 3 >= 2, 3 >= 3.

The *if* keyword introduces a selection statement.

```
var age;
...
if (age > 60)
  write("time to retire");
else
  write("carry on working");
```

If the value stored in *age* is more than 60, then *time to retire* is selected for printing. If the value stored in *age* is not less than 60, then *carry on working* is selected for printing.

The general format is

if (booleanExpression is true)
* statementSequence1*
else
* statementSequence2*

You need braces if a statement sequence contains more than one statement.

```
var invoiceValue, percentageDiscount, postage;
...
if (invoiceValue > 100.00) {
  percentageDiscount = 10.0;
  postage = 0.0;
}
```

The *else* clause is optional. You do not include it if you do not need it.

The human eye easily misses an else at the end of a long statement sequence. So, in general you try to arrange for the longest statement sequence to follow the *else*.

```
if (invoiceValue <= 0.0)
  write("error: invoice value cannot be zero or negative");
else {
  percentageDiscount = 0;
  postage = 1.50;
}
```

You want to make a selection from a set of many options. You could use a chain of *if else if ...* statements. You offer 10% for invoice values of between 100.01 and 199.99, 15% for invoice values between 200.00 and 400.00, and 25% for values more than 400.00.

% Discount	**Invoice Value**
10	100.01 - 199.99
15	200.00 - 400.00
25	> 400.00

You write

```
if (invoiceValue <= 100.00)
  percentageDiscount = 0.0
else if (invoiceValue < 200.00)
  percentageDiscount = 10.0;
else if (invoiceValue <= 400.00)
  percentageDiscount = 15.0;
else
  percentageDiscount = 25.0;
```

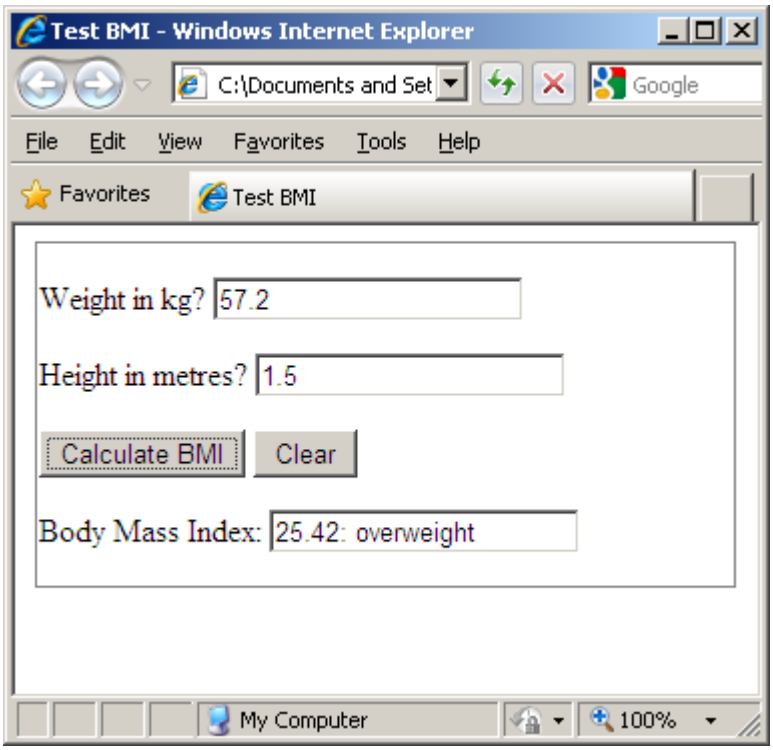
Understanding a chain of *else if*s is easy. You look down the sequence of Boolean expressions *(invoiceValue <= 100.00)*, *(invoiceValue < 200.00)*, *(invoiceValue <= 400.00)*, ...for the first one that is true and execute its corresponding statement block, then you skip to the end of the *if ... else if ...* structure. For example, suppose the *invoiceValue* was 250.00. The first Boolean expression that is true is *(invoiceValue <= 400.00)*. So you set *percentageDiscount* to 15.0 and finish. The trailing *else* deals with none of the above cases. The next program illustrates the point.

A person is clinically obese if their weight (in kg) divided by their height (in metres) squared >= 30.0, i.e. if their body mass index (bmi) is 30 or more.

```
isObese = (weight / (height * height) >= 30.0);
```

Our next JavaScript program inputs a weight and a height, and outputs a clinical weight state.

weight state	**bmi**
starvation	< 15.0
underweight	15.0 - 18.5
normal	18.6 - 24.9
overweight	25.0 - 29.9
obese	30.0 - 40.0
morbidly obese	> 40.0



Here is the HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Test BMI</title>
</head>

<body>
<div id="content">
<form name="frmBMI">
<fieldset>

<p>
<label for="weight">Weight in kg? </label>
<input type="text" name="weight">
</p>
```

```
<p>
<label for="height">Height in metres? </label>
<input type="text" name="height">
</p>

<p>
<input type="button" name="btnCalculate" value="Calculate BMI"
       onclick="calcBMI()">
<input type="reset" name="btnClear" value="Clear">
</p>

<p>
<label for="result">Body Mass Index: </label>
<input type="text" name="result">
</p>
</fieldset>
</form>
</div>
<script src="bmi.js"></script>
</body>
</html>
```

We use an HTML form to get the input from the visitor and to display the results. A call to the JavaScript function *calcBMI()* is made when the *Calculate BMI* button is clicked.

```
<input type="button" name="btnCalculate" value="Calculate BMI"
      onclick="calcBMI()">
```

We use JavaScript to analyse the results.

```
/* bmi.js: body mass index */

function calcBMI()
{
  var weight = parseFloat(document.frmBMI.weight.value);
  var height = parseFloat(document.frmBMI.height.value);
  var bmi = weight / (height * height);
  var report;

  if (bmi < 15.0)
    report = "starvation";
  else if (bmi < 18.6)
    report = "underweight";
  else if (bmi < 25.0)
    report = "normal";
  else if (bmi < 30.0)
    report = "overweight";
  else if (bmi <= 40.0)
    report = "obese";
  else
    report = "morbidly obese";

  document.frmBMI.result.value = bmi.toFixed(2) + ": " +
                                 report;
}
```

parseFloat() converts its string argument to a floating-point number - if it can. There is a similar JavaScript function *parseInt()* that converts its string argument to an integer.

The expression

```
document.frmBMI.weight.value
```

retrieves the value of the input item named *weight* in the form named *frmBMI* in the HTML document.

The statement

```
document.frmBMI.result.value = bmi.toFixed(2) + ": " + report;
```

copies the value of *bmi.toFixed(2) + ": " + report* into the item named *result* in the form named *frmBMI* in the HTML document.

We should really choose test values so that each boundary point (each Boolean expression) is exercised, but we shan't bother.

We also remember that arithmetic and comparisons involving float values are not always exact. We should also check that a valid range of numeric values has been entered. We acknowledge the problems and move on.

Exercise 1.3

1. Normal body temperature is 35.7 ± 2.5 Celsius i.e. between 33.2 and 38.2 inclusive. Invite the visitor to enter their body temperature. If it is less than 33.2 display *hypothermic*. If it is more than 38.2 display *hyperthermic*. If it is between 33.2 and 38.2 inclusive, display *normal*.

Now we look at the equality operators.

==	is-the-same-as
===	strictly-is-the-same-as
!=	not-the-same-as
!==	strictly-not-the-same-as

According to JavaScript, each of the following expressions is true.

```
false == 0
"" == 0
"5" == 5
```

But each of the following is false.

```
false === 0
"" === 0
"5" === 5
```

With *"5" == 5* JavaScript converts one type to the other. But with *"5" === 5* no automatic type conversion occurs.

We have the usual logical operators:

&&	and-at-the-same-time
||	either-one-or the-other-or-both

We can use the logical operators to define a valid range. For example, we might say that a valid age is one between 0 and 120 inclusive. We could write

```
var age;
var report;
...
if (age >= 0 && age <= 120)
  report = "age is valid";
else
  report = "age is invalid";
```

If an *age* value is zero or more, and at the same time, is 120 or less, i.e. between 0 and 120 inclusive, then it is valid, otherwise it is invalid.

Or we could write

```
if (age < 0 || age > 120)
  report = "age is invalid";
else
  report = "age is valid";
```

If an *age* is less than zero or more than 120, then it is invalid, otherwise it is valid.

Exercise 1.4

1. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years, but years divisible by 400 are. Design, write and test a script that inputs a year and outputs whether it is a leap year. You can determine whether a number is exactly divisible by 4, for example, if its remainder after dividing by 4 is zero. i.e. if (year % 4 === 0) ...

1.5 Arrays

The first five tracks on my play list are:

playList

0	Crown Imperial
1	Triple Crown
2	Les Huguenots
3	Grenadiers' Slow March
4	Brigade Quick Marches

An array is an indexed collection of elements. Indexing always starts from zero and the index numbers are contiguous (i.e. no gaps between any of them).

Each element of an array can contain a value. Here, array element 0 refers to *Crown Imperial*. And array element 4 refers to *Brigade Quick Marches*. Array element 5 is undefined - it does not exist.

We can declare an array named *playList* that contains just five tracks by writing

```
var playList = Array(5);
```

Array is a JavaScript word. And yes, the brackets must be round ones.

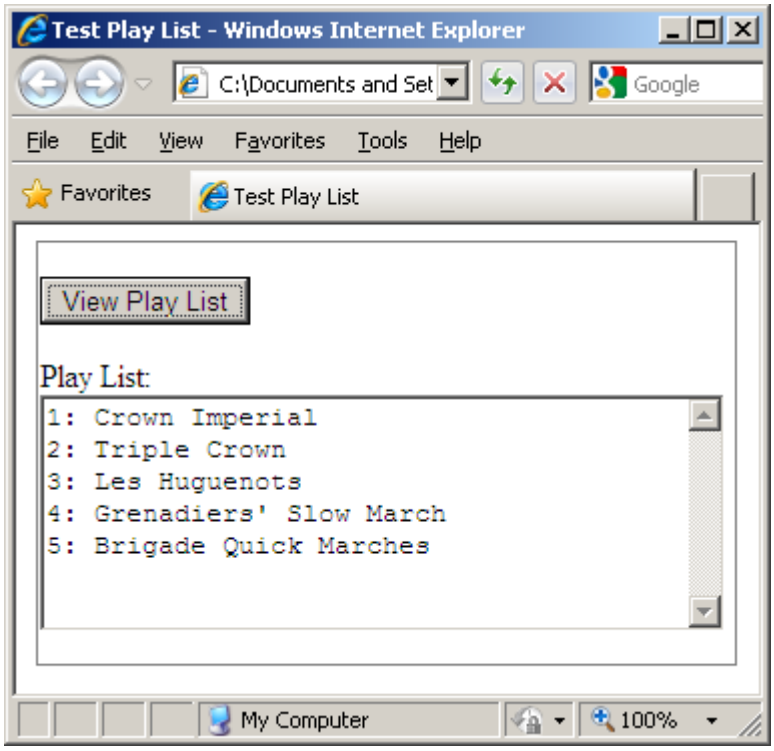
If a play list has an indeterminate number of tracks we can write

```
var playList = Array();
```

In JavaScript, the values of an array do not need to be all of the same type.

1.6 Iterations

Perhaps the most useful repetition construct is the *for ...* loop. We use a for loop to visit each element in an array in turn. Here is the output from our next script. The play list is displayed when the *View Play List* button is clicked.



The JavaScript is shown below.

```
/* playlist.js: array of music tracks */

/* showPlayList: fills an array with values and displays them */
function showPlayList()
{
  var aList = Array(5);
  var string = "";

  aList[0] = "Crown Imperial";
  aList[1] = "Triple Crown";
  aList[2] = "Les Huguenots";
  aList[3] = "Grenadiers' Slow March";
  aList[4] = "Brigade Quick Marches";
```

```
  for (var i = 0; i < aList.length; i++)
    string += (i + 1) + ": " + aList[i] + "\n";

  document.frmPlayList.list.value = string;
}
```

We declare an array named *aList* to have five elements, indexed from zero up to four. We use string to accumulate the contents of the array.

Each element of the array is given its value with

```
aList[0] = "Crown Imperial";
aList[1] = "Triple Crown";
aList[2] = "Les Huguenots";
aList[3] = "Grenadiers' Slow March";
aList[4] = "Brigade Quick Marches";
```

Then comes a *for ...* loop.

```
for (var i = 0; i < aList.length; i++)
  string += (i + 1) + ": " + aList[i] + "\n";
```

The variable *i* is used to refer to each element of the array. We say it indexes the array. Initially, its value is zero. We loop while *i* remains less than *aList.length*. JavaScript very conveniently provides the number of elements in an array with the *length* property.

For each iteration, every time round the loop, we add to *string*. We add the value of (i + 1), a colon, the contents of the i'th element of the *aList* array, and the new line character represented by *"\n"*. The last step in each iteration is to increment (add 1 to) *i*.

So, when

i = 0, we add 1: Crown Imperial (new line)
i = 1 we add 2: Triple Crown (new line)
i = 2 we add 3: les Huguenots (new line)
...

to the ever expanding string.

When *i* reaches 5, *(i < aListlength)* becomes false and the loop terminates.

In the HTML we use a form to display the button and the text area for the list.

```
<form name="frmPlayList">
<fieldset>
<p>
<input type="button" name="btnPlayList"
       value="View Play List"
       onclick="showPlayList()">
</p>
<p>
<label for="list">Play List: </label>
<textarea cols="40" rows="7" name="list"></textarea>
</p>
</fieldset>
</form>
```

When the button is clicked, the JavaScript function *showPlayList()* is called.

And you recall the line in the JavaScript

```
document.frmPlayList.list.value = string;
```

document is this HTML page. *frmPlayList* is the name of the form. *list* is the name of the text area.

Here is the entire HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Test Play List</title>
</head>

<body>

<div id="content">
<form name="frmPlayList">
<fieldset>
<p>
<input type="button" name="btnPlayList"
       value="View Play List"
       onclick="showPlayList()">
</p>
<p>
<label for="list">Play List: </label>
<textarea cols="40" rows="7" name="list"></textarea>
</p>
</fieldset>
</form>
</div>

<script src="playlist.js"></script>

</body>
</html>
```

Exercise 1.5

1. Use a JavaScript array to hold the number of days in each month (assume a non-leap year). Use the array to determine the day-in-year number when given a day in 1..7 and a month in 1..12.

1.7 Functions

We have written several functions in this tutorial, *write()*, *calcBMI()* and *showPlayList()* for example, without saying much about the purpose of functions and how they are written.

We write functions to perform small useful tasks. Doing so helps us to minimise errors since it easier to get a small simple thing right than a large complex one. Functions also helps us to manage size and complexity because we can focus on one small part of a script while

ignoring the rest. Functions helps us to minimise effort since we can re-use the same useful function in different situations.

Some functions are shown below.

```
/* min: returns the least of two numbers */
function min(x, y)
{
  if (x < y)
    return x;
  else
    return y;
}
```

The JavaScript word *function* introduces a function. Here, the function's name is *min* and it has two parameters named *x* and *y*. Parameters act like local variables to the function. We intend that both x and y are numeric. The body of the function is contained within braces. The value returned by a function can be used when the function is called and given actual argument values.

```
var least = min(2, 3);
```

The value returned by a function can be ignored. But that would be a bit pointless in our example. The function call can be written anywhere in the JavaScript file.

A function may have no parameters. And it is not necessary for a function to explicitly return a value.

```
/* showPlayList: fills an array with values and dislays them */
function showPlayList()
{
  var aList = Array(5);
  var string = "";

  aList[0] = "Crown Imperial";
  aList[1] = "Triple Crown";
  aList[2] = "Les Huguenots";
  aList[3] = "Grenadiers' Slow March";
  aList[4] = "Brigade Quick Marches";


  for (var i = 0; i < aList.length; i++)
    string += (i + 1) + ": " + aList[i] + "\n";

  document.frmPlayList.list.value = string;
}
```

An anonymous function has no name.

```
window.onload = function()
{
  var fahr = 98.4;
  var celsius = (5.0 / 9.0) * (fahr - 32.0);

  write(fahr + " degrees fahrenheit is " +
        celsius.toFixed(1) + " degrees celsius");
}
```

Now we look at the concepts of cohesion and coupling.

Cohesion refers to the number of tasks a function performs. A function with the highest level of cohesion performs the smallest task.

Coupling refers to the amount of data passed between functions. Functions communicate with each other via arguments and parameters. A pair of functions that share the smallest amount of data has the least coupling.

We seek to maximise cohesion and to minimise coupling. Why? If an error is found in a script its cause can usually be traced to just one small function, which can then be corrected without affecting other functions.

We shall use functions throughout chapter two, The DOM.

Bibliography

Keith J *DOM Scripting: Web Design with JavaScript and the Document Object Model* Apress 2005
Haverbeke M *Eloquent JavaScript* William Pollock 2011
www.tutorialpoint.com/javascript accessed April 2011