Introduction to JavaScript

This workbook provides a basic introduction to the JavaScript programming language. It includes explanations, code descriptions and code examples, which you should read. It also includes activity sections which ask you to carry out a set of actions, for example to create or modify some code. The activities reinforce what you have read, but also give you vital coding practice. The workbook is design to be read in sequence, if you skip parts it may make it harder to understand later parts.

Whilst the workbook includes everything you need to cover, you may find it useful to refer to other sources of information. The w3Schools website has a course on JavaScript, as does the codeacademy website. Our library also has books on JavaScript.

JavaScript

JavaScript is a high-level programming language. Its main area of application is the World Wide Web where it is used to add interactivity to web pages. HTML is used to specify the logical structure and semantics of the content, CSS is used to specify the presentation of the content as a web page, and JavaScript (JS) is used to specify the interactive behaviour of a web page. This can be thought of in terms of Progressive Enhancement. Progressive Enhancement is a web development strategy in which layers of "enhancement" are added to the base content. The "base" layer presents a complete usable solution, additional layers provide enhanced presentation, enhanced functionality and so on. In this way, the solution works regardless of the user's browser, device or bandwidth. Maximum accessibility is achieved.

In a simple model of Progressive Enhancement, there are three layers:

HTML

HTML + CSS

HTML + CSS + JS

For further detail and a more sophisticated model see Gustafson (2016).

Terminology Refresher

First it is useful to recall some basic HTML and CSS terminology as this will aid in our understanding subsequent explanations and examples.

HTML Elements

An HTML page consists of a number of HTML elements. An HTML element usually consists of a Start Tag, some Content, and an End Tag. The Start Tag may include one or more Attributes, each consisting of an Attribute Name and an Attribute Value.

```
<p>Hello World</p>
```

	Start Tag
Hello World	Content
	End Tag

```
<p class="greeting">Hello World</p>
```

	Start tag

class="greeting"	Attribute
class	Attribute Name
greeting	Attribute Value
Hello World	Content
</p>	End tag

There are also a small number of empty HTML elements, which consist of just a Start Tag, with no Content, and no End Tag. The Start Tag may include one or more Attributes, each consisting of an Attribute Name and an Attribute Value.

```
<img src="face.png">
```

<img>	Start Tag
src="face.png"	Attribute
src	Attribute Name
face.png	Attribute Value

CSS Rules

A Cascading Style Sheet consists of a number of CSS Rules. A CSS Rule consists of one or more Selectors and a Declaration Block containing one or more Declarations. Each Declaration consists of a Property Name and a Property Value.

```
.greeting {color: yellow;}
```

.greeting	Selector
{color: yellow;}	Declaration Block
Color: yellow;	Declaration
color	Property Name
yellow	Property Value

Multiple selectors can be specified:

`li, b {…}`
The Declarations in the Declaration Block will be applied to <li> Tags and to <b> Tags

Selectors can be nested:

`li b {…}`

The Declarations in the Declaration Block will be applied to <b> Tags which occur within a <li> Tag. For example:

```
<ol>
      <li>This is very <b>important!</b></li>
      …
</ol>
```

Notice the important difference a comma can make when indicating multiple or nested Selectors.

Selectors can also refer to a Class:

```
.greeting {…}
```

The Declarations in the Declaration Block will be applied to all Tags where the `class` Attribute has the Value `greeting`. For example:

```
<p class="greeting">Hello World</p>
```

Note the use of the full stop in the Selector to indicate that this Declaration Block applies to a Class rather than a Tag.

Selectors can also refer to an ID (an identifier):

```
#headline {…}
```

The Declarations in the Declaration Block will apply to the single Tag where the `id` Attribute has the Value `headline`. For example:

```
<h1 id="headline">Breaking News</h1>
```

Note, a particular `id` Value can only be used once in an HTML document, it is a unique identifier. It can however be used by other HTML documents which use the same external CSS.

Connecting JavaScript and HTML

There are three ways in which we can connect JavaScript to our HTML. These are very similar to the ways in which we can connect to CSS style rules.

First we can include the JavaScript statements inline with the HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Inline Example</title>
</head>
<body>
    <h1>A First Example</h1>
    <p><script>document.write('Hello World');</script></p>
</body>
</html>
```

In this example we can see the JavaScript statement between the `<script>` start and end tags. The script is executed at the point it appears in the HTML document.

Activity

1. Create a new HTML file, example1.htm, containing the above code.
2. Upload it to the student webserver.

3. View the page in a browser. Note that the `document.write` command has inserted the string Hello World into the document.
4. View the source of the page. Notice that the source hasn't been amended by the script, only the output has been changed. The browser has interpreted the script (and the HTML tags, and CSS rules if we had any) as it renders (displays) the page.
5. Validate the page using an HTML validation service. The page is still valid HTML. It is important to note that the HTML validation service will not detect bugs in your JavaScript.

Second, we can define a JavaScript function within the HTML and then call it:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Embedded Procedure Example</title>
    <script>
    function sayHelloWorld() {
        document.write('Hello World');
        }
    </script>
</head>
<body>
    <h1>A Second Example</h1>
    <p><script>sayHelloWorld();</script></p>
</body>
</html>
```

In this example we can see a JavaScript function, `sayHelloWorld()`, defined in the <head> of the document between the <script> tags. This function is then called from within the <body> of the document, with the function call between <script> tags.

Activity

6. Create a new HTML file, example2.htm, containing the above code.
7. Upload it to the student webserver.
8. View the page in a browser. The output is the same as example1.
9. View the source of the page. Again notice that the source hasn't been amended by the script, only the output has been changed.
10. Validate the page using an HTML validation service. The page is still valid HTML. It is important to check that we haven't broken our HTML when we are focussed on our JavaScript.

However, neither of these approaches is the preferred one. Just like with CSS, the preferred option is to keep the JavaScript in a separate file and link to it from the HTML file. We can then call the JavaScript functions from our HTML as and when we need them.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS External Procedure Example</title>
    <script src="example.js"></script>
</head>
<body>
    <h1>A Third Example</h1>
    <p><script>sayHelloWorld();</script></p>
</body>
</html>
```

In this example we can see that the <head> now contains a link to a JavaScript file called example.js. A JavaScript file is a plain text file, just like HTML and CSS. It can be written in any text editor (such as Notepad, Notepad++ or Atom). A JavaScript file uses a .js file extension. The example.js file would contain the required function definition:

```
function sayHelloWorld() {
    document.write('Hello World');
}
```

The function is then called from within the <body> of the document, with the function call between <script> tags.

Activity

11. Create a new HTML file, example3.htm, containing the above HTML code.
12. Create a new JavaScript file, example.js, containing the above JavaScript.
13. Upload both files to the student webserver.
14. View the HTML page in a browser.
15. View the source of the page. Again notice that the source hasn't been amended by the script, only the output has been changed. The browser has used the function stored in the linked JavaScript file, example.js, in the same way that it might use CSS Rules in a linked CSS file.

We have used the document.write command in the above examples, and we have seen its effect on the HTML page, but what does it actually mean? A web browser makes use of two important Objects when rendering HTML pages, the first is a Window Object which is used to represent each window or tab. The second is the Document Object which represents the actual page. This is created from the HTML, CSS and JS files associated with the page. This Document Object has Properties associated with it, such as a Title and a URL. It has Methods, which allow us to retrieve information from or to modify the document. It also has Events, such as a user clicking on the page, which can act as triggers for actions (such as executing a Method). This Document Object is rendered by the browser.

```
document.write('Hello World');
```

`document`	Document Object
`.`	Member Operator
`write()`	Method
`'Hello World'`	Parameters
`;`	Don't forget the ;

So, this statement uses the `write()` method belonging to the `Document` Object. This method allows new content to be written into the rendering page at the point where the script occurs. The browser is manipulating the Document Object, not the underlying HTML file, this is why the HTML source of the page is unchanged.

If we look at the example.js file we created earlier

```
function sayHelloWorld() {
    document.write('Hello World');
}
```

`function`	Function keyword
`sayHelloWorld()`	Name of the function
`{`	Start of the Code Block
`document.write('Hello World');`	Statements in the Code Block
`}`	End of Code Block
	No ; here!

We are defining a function (using the `function` keyword). In other programming languages you may have used, these might have been called routines, subroutines, procedures or subprograms.

We have named our function sayHelloWorld and indicated that it has no input parameters () – this means we pass no data values to it when we call it.

The function comprises a single Code Block containing the statements which are executed when the function is called. Each statement is terminated by a semi-colon ;

Functions can also return data values, but this one does not.

Activity

16. Add a new function called sayGoodbyeWorld() to example.js which writes the message 'Goodbye World' using the document.write() method
17. Add a new paragraph to example3.htm
18. Include a call to the sayGoodbyeWorld() function from within this new paragraph.
19. Upload both files to the student webserver.
20. View the HTML page in a browser.

As when programming in any language, we should ensure that our code is well laid out using indentation and alignment appropriately. We should also comment our code. JavaScript allows both multiline and single line comments:

```
/* This function displays a message.
    It is not very complicated */

function sayHelloWorld() {
    document.write('Hello World'); // writes the message
}
```

While commenting such a short piece of code isn't really necessary, it is a good habit to get into straight away.

Multiline comments are also an effective way of preventing parts of the code running during debugging or testing.

The following function provides an answer to that age old student question, "if I got 55% in my coursework and the coursework contributes 50% of the module marks, how many marks have I got for the module so far?"

```
/* This function calculates the contribution of a
    Coursework mark to the overall module mark */

function calculateModulePercent() {
    var courseworkMark; // mark in the coursework
    var courseworkContribution; // contribution to the module
    var modulePercent; // contribution to overall mark

    courseworkMark = 55;
    courseworkContribution = 50;
    modulePercent = courseworkMark * (courseworkContribution / 100);

    document.write(modulePercent);
}
```

This example introduces a couple of new syntax items.

Firstly variables. Let's just take for granted that we know what variables are and that all we need to know is how to use them in JavaScript.

```
Var courseworkMark;
```

var	Variable keyword
courseworkMark	Name of the variable
;	Don't forget the ;

So, this defines a variable named courseworkMark, done, simple. You may find this entirely satisfactory… or you may be thinking "OK, but what type of variable, does it hold an integer, or a

string or what?" JavaScript uses "untyped" variables, it doesn't care what sort of data you store in courseworkMarks, it could be a number, it could be a string, it could be a Boolean, or even something more complicated. Just to really irritate people who are used to strongly typed variables, you could store a number in it now, then later store a string in it and then later on maybe a Boolean. This is all fine in JavaScript.

However, as a general rule it is better to have well-named variables that serve a clear purpose in your script than to keep reusing them for different purposes.

```
courseworkMark = 55;
```

courseworkMark	Name of the variable
=	Assignment operator
55	Value to be assigned
;	Don't forget the ;

This sets the value of the courseworkMark variable to the number 55. Note carefully that = is the assignment operator, it assigns the value 55 to the variable. It is not testing for equality.

Alternatively, we could assign a string to the variable:

```
courseworkMark = 'excellent';
```

or a Boolean:

```
courseworkMark = false;
```

Note that this:

```
courseworkMark = '55';
```

assigns a string to the variable, not a number. This is one to keep an eye on as strings behave differently to numbers. For example, consider:

```
var itemOne = '55';
var itemTwo = '45';
var total = itemOne + itemTwo;
```

Both variables hold strings, when you add strings, they concatenate (join together), so total would hold the string '5545'.

If we had:

```
var itemOne = 55;
var itemTwo = 45;
var total = itemOne + itemTwo;
```

Both variables hold numbers, so they add as normal, so total would hold the number 100.

Finally, just to really disturb people who are used to strongly typed variables we can do this:

```
var itemOne = 55;
```

```
var itemTwo = '45';
var total = itemOne + itemTwo;
```

Yes, "adding" a number and a string! JavaScript treats both as strings and concatenates them, so total would hold the string '5545'. This only works for the addition operator, +, the other operators just return a value of NaN (not a number).

Activity

21. Create a new JavaScript file containing the calculateModulePercent() function.
22. Create a new HTML page which displays the result of the calculateModulePercent() function.
23. Vary the values of courseworkMark and courseworkContribution to confirm that the function works correctly.

Currently the way we are writing to the document is rather unsophisticated, we are just writing wherever the function is called from in the HTML. Let's look at doing something a little more interesting.

```
/* This function calculates the contribution of a
    Coursework mark to the overall module mark */

function calculateModulePercent() {
    var courseworkMark; // mark in the coursework
    var courseworkContribution; // contribution to the module
    var modulePercent; // contribution to overall mark

    courseworkMark = 55;
    courseworkContribution = 50;
    modulePercent = courseworkMark * (courseworkContribution / 100);

    var el = document.getElementById('score');
    el.textContent = modulePercent;
}
```

You will notice in the above example that we are not using the `document.write()` method. Instead, we are writing content into a specific element in the Document Object Model (DOM). We will look at the DOM and ways to interact with it in more detail later. For now, we will just consider this example.

```
var el = document.getElementById('score');
```

`var`	Variable keyword
`el`	Name of the variable
`=`	Assignment operator
`document`	Document Object

`.`	Method Operator
`getElementById()`	Method
`'score'`	Parameter
`;`	Don't forget the ;

The `getElementById()` method returns an element from the Document Object Model whose id is equal to the parameter it has been passed, in this case the string 'score'. This element is defined by an HTML start Tag and an end Tag (unless it is an empty Tag, in which case it is just the start Tag). So, if our HTML document contained `<p id="score">Unable to display score</p>` this element would be returned as its id is equal to 'score'. Once we have this element in a variable, we can manipulate it.

```
el.textContent = modulePercent;
```

`el`	Name of the variable
`.`	Method Operator
`textContent`	Element property
`=`	Assignment operator
`modulePercent`	Variable
`;`	Don't forget the ;

This sets the `textContent` property of whatever is in the variable `el` (in this case an element with the id = 'score') to the value of the variable `modulePercent`. So, if our HTML document contained `<p id="score">Unable to display score</p>` the `textContent` of the element (`Unable to display score`) will be changed to whatever is in the variable `modulePercent`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Mark Feedback</title>
    <script src="newexample.js"></script>
</head>
<body>
    <h1>Your Mark</h1>
    <p id="score">Unable to display score</p>
    <p id="message">Unable to display message</p>
    <script>calculateModulePercent();</script>
</body>
</html>
```

Activity

24. Create a new HTML page containing the above HTML
25. View the HTML page in a browser. We have not created newexample.js yet, so the content of the two paragraphs will be rendered.

26. Create a new JavaScript file called newexample.js containing the calculateModulePercent() function and the new getElementByID method as shown above.
27. View the HTML page in a browser. The content of the first paragraph should now be the value of modulePercent.
28. View the source of the HTML page, again note that the source is unchanged, it is the DOM that has been modified by the JavaScript.

Let's add a second function to the JavaScript file to display an appropriate message based on the coursework mark.

```
/* This function creates an appropriate message
    based on the coursework mark */

function createMessage(studentMark) {
    var studentName = 'Chris'; // Student name
    var studentMark; // coursework mark
    var message; // message to student

    if (studentMark<40) {
        message = 'Unlucky ' + studentName;
    }
    else {
        message = 'Good job ' + studentName;
    }

    var el = document.getElementById('message');
    el.textContent = message;
}
```

This example introduces a couple of different things. Firstly we have a function with a parameter – this function receives data from the function which calls it. When we call it we need to pass a parameter value, so if we called it from within `calculateModulePercent()` we could pass the value of the variable `courseworkMark` in which case the call would be `createMessage(courseworkMark);`

`function createMessage(studentMark) {…}`

`function`	Function keyword
`createMessage()`	Function name
`studentMark`	Parameter
`{…}`	Function code block

We can pass multiple parameters to a function, the parameters are separated by commas, e.g. `function createMessage(studentName, studentMark) {…}` . The parameters can be numbers, strings, Booleans, or more complex data types.

The example also includes conditional statements (if statements). A conditional statement consists of the `if` keyword, a condition in parentheses and a code block. If the condition evaluates to true, the code block is executed.

```
if (studentMark<40) {
        message = 'Unlucky ' + studentName;
    }
```

`if`	If keyword
`(studentMark<40)`	Condition
`{…}`	Code block
	No ; here

We can add an `else` keyword and a second code block that is executed if the condition evaluates to false.

```
if (studentMark<40) {
        message = 'Unlucky ' + studentName;
}
else {
        message = 'Good job ' + studentName;
}
```

`else`	else keyword
`{…}`	Code block
	No ; here

We can also add alternative conditions.

```
if (studentMark<40) {
        message = 'Unlucky ' + studentName;
}
else if (studentMark<50) {
    message = 'Not bad ' + studentName;
}
else {
        message = 'Good job ' + studentName;
}
```

These conditions are evaluated in sequence until one of them evaluates to true, or the `else` statement is reached.

Activity

29. Add the `createMessage()` function to newexample.js
30. Add a function call to the `createMessage()` function from the `calculateModulePercent()` function. Remember to pass the parameter.
31. View the HTML page in a browser.
32. Update `createMessage()` so that the messages are 'Fail' (<40), 'Pass' (40-49), 'Good Pass' (50-59), 'Merit' (60-69) and 'Distinction' (70-100). Any other value should display the message 'Error'.
33. Check that all the messages work correctly by varying the value of `courseworkMark`.

Using multiple `else if` statements in a conditional is inefficient due to the way it is processed (all the conditions are checked even if one has already evaluated to true). In some cases, we can use a `switch` statement instead, which is more efficient.

```
switch(switchValue) {
    case value1:
        statementA;
        statementB;
        break;
    case value2:
        statementC;
    case value3:
        statementD;
        break;
    default:
        statementE;
        break;
}
```

`switch`	Switch keyword
`switchValue`	A variable or expression
`case`	Case keyword
`:`	Note – this is a colon :
`value1`	Value to compare to switchValue
`statementA`	Statement to be executed
`;`	Remember the semi-colon ;
`break`	Break keyword, exits the switch
…	
`default`	The default case

The switch() function takes a single variable, or an expression. This is then compared to the value associated with one or more cases. If the variable or expression is equal to the case value, then the statements associated with the case are executed. A break keyword is used to exit the switch statement. The default case is executed if no case value has been matched.

It is important to note that once a case value has been matched, all following statements (including those in other cases) will be executed until the switch is exited using a break. So in the above example:

case	executes	then executes
switchValue = value1	statementA	statementB
switchValue = value2	statementC	**statementD**
switchValue = value3	statementD	
None of the above (default)	statementE	

Note that because case value2 has no `break`, it executes statementD as well as statementC. This can be useful, but missing out a break statement accidentally can lead to confusing errors.

Activity

34. Write a new function `levelCourse()` which uses a switch statement to display an appropriate message depending on the level of the course. Level 4 should display 'first year', level 5, 'second year' and level 6, 'final year'. Any other value should display 'You are not on a degree course'.
35. Create an appropriate HTML page to use the function.
36. Test the function with different values to ensure it performs correctly.

Often we will want to use data structures that are more sophisticated than simple strings or numbers. An array allows us to store a list of related values, without necessarily knowing how many values we will need to hold. Consider the following example.

```
var modules;
modules = ['Maths', 'Programming', 'Web Development'];
```

We have defined a variable called modules and assigned an array to it. The array is identified using the square braces […]. We have put three values into the array, in this case three strings. The values are separated by commas. We can refer to specific values in the array by using their index number. Note: the values are indexed starting at zero. So, `modules[0]` is the string `'Maths'` and `modules[2]` is the string `'Web Development'`. We can use this to access values in an array and also to change them. Consider the following example.

```
var modules;
var oldModule;
var newModule = 'Graphics';

modules = ['Maths', 'Programming', 'Web Development'];

oldModule = modules[1];
modules[1] = newModule;
```

If we ran this code then `oldmodule` would have the value `'Programming'` and `modules` would have the value `['Maths', 'Graphics', 'Web Development']`.

Arrays have a number of properties and methods associated with them, including.

`.length` returns the number of values in the array.

`.pop()` removes the last value from an array and returns that value.

`.push(value/s)` adds the value or values to the end of the array and returns the new array length.

`.shift()` removes the first value from an array and returns that value.

`.unshift(value/s)` adds the value or values to the start of the array and returns the new array length.

There are others, but you can look those up yourself.

One of the things we will often want to do with arrays is to move through them value by value, performing some operation on each value. One way we can do this is using a `for` loop.

```
function showModules() {
    var modules = ['Maths', 'Programming', 'Web Development'];
    var arrayLength = modules.length;
    var modulesMessage = '';
    var i;

    for (i = 0; i < arrayLength; i++) {
        modulesMessage = modulesMessage + modules[i] + '<br>';
        }
}
```

A `for` loop uses a counter and a condition to control how many times a block of code is executed.

```
for (i = 0; i < arrayLength; i++) {
  …
 }
```

`for`	for keyword
`()`	Contains the loop parameters
`i = 0`	Initial counter state
`i < arrayLength`	The condition
`i++`	Update counter at end of loop
`{…}`	Code block to be executed
`;`	Note the semi-colons separating the loop parameters

In a `for` loop the counter is set to its initial value. The condition is then checked. If the condition is true, then the code block will run. If the condition is false then the code block will not run and the loop will exit. If the code block was run, then the counter is updated and the condition is checked again.

So, in the code above, we have initialised our counter, `i`, to zero. This is because we are going to use the counter to index the values of the array, and the values of an array are indexed starting from zero. The condition is that `i` is less than the length of the array (`modules.length`) – the length of the array in this case is 3, and its values are `modules[0]`, `modules[1]` and `modules[2]`, so we only want to execute the code block when `i` is less than the length of the array. Our update, `i++`, increases `i` by one, so we can step through the array value by value. Taken as a whole, the `for` loop steps through the values in the array concatenating the values together into a single string with each value followed by the string '
' representing a line break in HTML. So we might expect the result to be each of the modules shown on a separate line when we view the output in an HTML page.

Activity

37. Create a new JavaScript file and associated HTML file which implements the `showModules()` function shown above.
38. Use `.getElementById` and `.textContent` to write the resulting string to an HTML page.
39. Consider the resulting output, but not for too long…

…because it wasn't really what we wanted. Our `'<br>'` strings have been displayed as text content rather than being interpreted as HTML mark-up. The clue is in the name, `.textContent` – we have specified that the string is to be treated as text content. This is handy, but not what we intended in this case. If you replace `.textContent` with `.innerHTML` you will achieve the desired outcome. `.innerHTML` will treat text that looks like HTML mark-up as HTML mark-up. This is one way we can add HTML using JavaScript. We will look at some more elegant and powerful ways later.

Activity

40. Update the `showModules()` function to use .innerHTML instead of .textContent.
41. Confirm that it works as expected.
42. Rewrite the `showModules()` function so that it uses two arrays, `modules = ['Maths', 'Programming', 'Web Development']` and `marks = [45, 98, 65]`. The output displayed in the HTML page should show each module and its mark, as a percent, on a separate line, e.g. Maths 45%.
43. Rewrite the `showModules()` function again so that it uses a single array, `modules = ['Maths', 45, 'Programming', 98,'Web Development', 65]`. The output displayed in the HTML page should show each module and its mark, as a percent, on a separate line, e.g. Maths 45%.

`for` loops are pretty versatile, but there are other loops which we can use.

If this is our code as a `for` loop…

```
for (i = 0; i < arrayLength; i++) {…}
```

`i = 0`	Initial counter state
`i < arrayLength`	The condition
`i++`	Update counter at end of loop
`{…}`	Code block

…this is the same code as a `while` loop…

```
var i = 0;
while (i < arrayLength) {
    …
    i++;
    …
    }
```

`var i = 0`	Initial counter state
`i < arrayLength`	The condition
`i++`	Update counter during the code block
`{…}`	Code block

Note that the counter is initialised before the `while` statement and it is updated as part of the code block.

In this case the `for` loop and the `while` loop do exactly the same thing. Generally we would use a `for` loop when we want to step through a set of values and a `while` loop when we want to repeat a block of code until some condition is true.

The do `while` loop looks similar, but has an important difference… the condition is checked at the end of the loop not the beginning, so the code block will always execute at least once.

```
var i = 0;
do {
    …
    i++;
    …
    } while (i < arrayLength);
```

`var i = 0`	Initial counter state
`i < arrayLength`	The condition
`i++`	Update counter during the code block
`{…}`	Code block

This do while loop looks like it will do exactly the same as the for and while loops shown previously, but it won't in all cases! Suppose there are no items in the array, so arrayLength is zero, then the for loop and the while loop will not run the code block because they check the condition first and i is not less than zero, it is equal to zero. This means that the value of modulesMessage is still an empty string, ''.

The do while loop will run the code block once. The array is empty, so modules[0] has the value undefined. modulesMessage is concatenated with this to produce the string 'undefined'. The condition is then checked and is false, so it will not run it a second time. In this case, the value of modulesMessage is the string 'undefined'.

Typically we will use for or while loops, do while is less common.

Activity

44. Write a new version of the showModules() function so that it uses a while loop instead of a for loop.
45. Write a new version of the showModules() function so that it uses a do while loop instead of a for loop (even though we know it won't work correctly on empty arrays).
46. Test the for, while and do while versions of the showModules() function to verify that they behave in the same way on non-empty arrays, but differently on empty arrays.
47. Modify the do while version of the showModules() function to include an if or switch statement such that it behaves the same way as the for and while versions in the case of an empty array.

Earlier on we used an array that contained two different types of data, module names and module marks, ['Maths', 45, 'Programming', 98,'Web Development', 65]. We will often want to create and manipulate complex data structures which hold a number of different pieces of data. In JavaScript we can use objects which contain properties (variables) which can have values. These objects can also contain methods (functions) which allow the object to be manipulated or interrogated.

```
var student = new Object();

student.firstName = 'Chris';
student.lastName = 'Jones';
student.course = 'BSc Computing';
student.moduleMarks = ['Maths', 45, 'Programming',
98,'Web Development', 65];
```

new	Keyword
Object()	Constructor function for objects

This code create a new object called student. It adds a property called firstName to student and assigns the value 'Chris' to it. It adds a property called lastName to student and assigns the value 'Jones' to it. It adds a property called course to student and assigns the value 'BSc

Computing' to it. It adds a property called moduleMarks to student and assigns the array ['Maths', 45, 'Programming', 98,'Web Development', 65] to it.

Once we have created our object we can retrieve its property values, for example:

```
studentFullName = student.firstName + ' ' + student.lastName;
```

We can also change its property values, for example:

```
student.firstName = 'Christopher';
student.moduleMarks[1] = 77;
```

If we assign a value to a property that doesn't exist, that property will be added to the object. For example:

```
student.enrollmentNumber = 948464;
```

would add the property enrollmentNumber to student and assign the value 948464 to it.

We can remove a property from an object using the delete keyword, for example:

```
delete student.enrollmentNumber;
```

Note that this deletes the property and its value, it does not delete the entire object.

If we just want to clear the value of a property, rather than delete it, we can just set its value to undefined, for example:

```
student.course = undefined;
```

Alternatively we could set it to an empty string, so long as an empty string is not a valid value for the property.

Activity

48. Write a new version of the showModules() function that holds the student data in an object as shown above.
49. The output displayed in the HTML page should show the following:

    Student Name: Chris Jones
    Course: BSc Computing
    Results:
    Maths 45%
    Programming 98%
    Web Development 65%

At the moment we can only use our student object to hold a single student's details which is rather limiting. It might be more useful to create a reusable template for the student object which we could then use to create a new student object each time we needed one. First we create a function for producing a student object:

```
function Student(firstName, lastName, course, moduleMarks) {

    this.firstName = firstName;
    this.lastName = lastName;
```

```
    this.course = course;
    this.moduleMarks = moduleMarks;
}
```

Note the use of the `this` keyword to indicate that the property belongs to the object that *this* function creates. Typically we will use a capital letter at the beginning of object constructor function names to differentiate them from other functions.

Now that we have our function we can create new objects from it, using the new keyword, for example:

```
var student1 = new Student('Tom', 'Jenkins', 'BSc Computing',
['Maths', 55, 'Programming', 87,'Web Development', 65]);

var student2 = new Student('Mary', 'Evans', 'BSc Computer
Networking', []);
```

We can then access an objects values in the usual way, for example:

```
student2FullName = student2.firstName + ' ' + student2.lastName;
```

Objects can also contain methods (functions) which allow the object to be manipulated or interrogated. This is useful when there are actions which we will want to perform many times on an object or set of objects. In our example, we might want a method which returns the students full name. We can include a method to do this our object template:

```
function Student(firstName, lastName, course, moduleMarks) {

    this.firstName = firstName;
    this.lastName = lastName;
    this.course = course;
    this.moduleMarks = moduleMarks;

    this.fullName = function() {
        return this.firstName + ' ' + this.lastName;
    }
}
```

So, our Student object now has a method we can call, for example:

```
student2FullName = student2.fullName();
```

Activity

50. Write a new version of the `showModules()` function that uses the `student` object constructor as shown above. Create objects for Chris Jones, Tom Jenkins and Mary Evans using the data shown above.
51. The output displayed in the HTML page should show the following:

    Student Name: Chris Jones
    Course: BSc Computing
    Results:
    Maths 45%
    Programming 98%

Web Development 65%

Student Name: Tom Jenkins
Course: BSc Computing
Results:
Maths 55%
Programming 87%
Web Development 65%

Student Name: Mary Evans
Course: BSc Computer Networking
Results:
No results found

52. Add a new function `fullResults()` to your `student` object which returns a string containing the correctly formatted results for the student. Rewrite your `showModules()` function accordingly.

As we have seen earlier, we can store arrays in objects. We can also store objects in arrays.

At the moment each of our student objects is stored in a variable, but it is rather untidy to have to create a new variable every time we want to add a student, especially if we don't know how many students we will need to add. If we hold them in an array it is easy to add, delete, and modify them.

Adding objects to an array is simple, for example:

```
var studentArray = [];

studentArray[0] = new Student('Tom', 'Jenkins', 'BSc Computing',
['Maths', 55, 'Programming', 87,'Web Development', 65]);
```

We can then access properties, for example:

```
el.innerHTML = studentArray[0].course;
```

or:

```
el.innerHTML = studentArray[0].moduleMarks[2];
```

We can also access methods, for example:

```
el.innerHTML = studentArray[0].fullName();
```

Now that we have our students in an array we can use a loop to go through all the student records in turn.

Activity

53. Write a new version of the `showModules()` function that uses the `student` object constructor but stores the objects in an array, as shown above. Create objects for Chris Jones, Tom Jenkins and Mary Evans using the data shown earlier. Use a loop and the `fullName()` and `fullResults()` functions to generate the output.
54. The output displayed in the HTML page should show the following:

    Student Name: Chris Jones

Course: BSc Computing
Results:
Maths 45%
Programming 98%
Web Development 65%

Student Name: Tom Jenkins
Course: BSc Computing
Results:
Maths 55%
Programming 87%
Web Development 65%

Student Name: Mary Evans
Course: BSc Computer Networking
Results:
No results found

55. Add a new function `fullRecord()` to your `student` object which returns a string containing the complete correctly formatted student record (such as for Mary Evans above). Rewrite your `showModules()` function accordingly.

This workbook has provided a basic introduction to the JavaScript programming language. It has not covered all the details, so you may well need to do some additional reading or look up specific details when you need them.

This workbook has focussed mainly on the Javascript language itself, the relationship to HTML pages has been rather limited. In the next workbook, we will look in more detail at the Document Object Model (the DOM), and how we can manipulate it using JavaScript.

References

Gustafson, A. (2016). *Adaptive Web Design: Crafting Rich Experiences with Progressive Enhancement*. New Riders.