

JAVASCRIPT - THE DOM, ARRAYS, FORMS

JAVASCRIPT AND THE DOM

Javascript is a client side scripting language. In web development it is mainly used to manipulate the DOM (Document Object Model).

The DOM is the HTML tree that makes up the HTML page. Javascript can be used to identify elements in the DOM and then be used to amend them. This is done by either adding HTML or CSS to change the appearance of the page. Javascript can also be used to insert new DOM elements, ie add in new HTML elements not originally in the file, and if needed, it can also remove DOM elements.

ADDING JAVASCRIPT TO A WEB PAGE

When adding Javascript to a web page there is a similar range of options to those you are hopefully familiar with from CSS. In CSS, styles can be added inline (directly to HTML element), at a document level (using the `<style>` tag) or loaded from an external stylesheet most frequently with the `<link>` tag.

With Javascript it can be added inline such as:

```
<body onload="alert('Just a Test')">
```

However, this technique is generally not recommended (as with inline CSS) as it makes code difficult to manage.

Javascript can also be added on a document basis. This is done with the `<script>` tag. Inside of this can be added Javascript.

```
<script>
alert('Fired from the script');
</script>
```

Finally, Javascript can be held in an external JS file and referenced by the document that wishes to use it. This is also done using the `<script>` tag as follows:

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

The value of the `src` attribute is a path to a JS file in the file structure. This can be a relative or absolute path as necessary. Inside the `js/myScript.js` will just be Javascript.

The `<script>` tag can be added anywhere in the document. However, it is good practise to either:

1. Place the `<script>` at the top of the document as a child of the `<head>` of the HTML file.
2. Place the `<script>` near to the bottom of the document before the closing `</body>` tag.

Either approach is valid but if you do place the `<script>` at the top of the document you should ensure you code only runs when the document has loaded (more later).

ALERTS AND CONSOLES

Open the file `index.html`. Here is a standard HTML document with the usual mix of images and text.

Add the following before the closing `</head>` tag:

```
<script>
alert('Hello World');
</script>
```

Save and refresh you page in the browser. You should see a Javascript alert box.

Note: Javascript alert boxes are just simple dialogue boxes. Although tempting to use them to provide user feedback they are little untidy and often associated by users as 'errors'. It is much better for users to add messages to the DOM rather than as alerts.

Values in the DOM can be extracted using a range of methods and properties. Most of these relate to a Javascript object known as `document`. This represents an object that represents the DOM.

Change the alert above to the following:

```
<script>
console.dir(document);
</script>
```

In Google Chrome use the developers tools to locate the 'console'. This is an essential part of the toolkit when working with Javascript.

The `console.dir()` command in the above snippet tells the console to output the content of the document object.

```

▼ #document ⓘ
  URL: "http://homepages.shu.ac.uk/~cmsmjcdemos/js-dom-programming/#"
  ▶ activeElement: body
  ▶ alinkColor: ""
  ▶ all: HTMLAllCollection(84) [html, head, meta, title, meta, meta, meta, link, link,
  ▶ anchors: HTMLCollection []
  ▶ applets: HTMLCollection []
  ▶ baseURI: "http://homepages.shu.ac.uk/~cmsmjcdemos/js-dom-programming/#"
  ▶ bgColor: ""
  ▶ body: body
    ▶ characterSet: "UTF-8"
    ▶ charset: "UTF-8"
    ▶ childElementCount: 1
  ▶ childNodes: NodeList(2) [<!DOCTYPE html>, html]
  ▶ children: HTMLCollection [html]
    ▶ compatMode: "CSS1Compat"
    ▶ contentType: "text/html"
    ▶ cookie: "rxVisitor=149936675646378JQBR4TMMU3N5CBIDCATA0DAVPM0659; dtLatC=22; dtCook
    ▶ currentScript: null
  ▶ defaultView: Window {postMessage: f, blur: f, focus: f, close: f, frames: Window, ..
    ▶ designMode: "off"
    ▶ dir: ""
  ▶ doctype: <!DOCTYPE html>

```

When adding Javascript to your pages you will find the console very useful. As well as `console.dir()` which is used to debug objects you can also use other commands such as `console.info()` and `console.log()`.

Other browser do have similar tools but this author rates the Chrome tools as currently the most useful.

GETTING TO KNOW THE DOM

We saw above that there is a `document` object. This object has a large range of methods and properties. These can be used to target elements in the DOM, and then extract or change their value.

The easiest of the many methods used to identify DOM elements looks for an element by id and is called `document.getElementById()`.

Once an element is targeted we could retrieve its contents using the property `innerHTML`.

Notice that the `<h1>` has an id attribute:

```
<h1 id="myHeading">My Demo Site</h1>
```

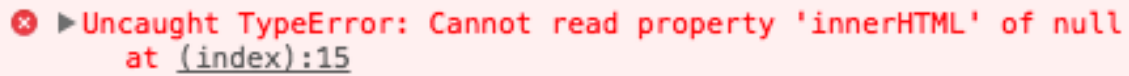
To retrieve the value inside of this `<h2>` we can use:

```
document.getElementById("myHeading").innerHTML
```

We could try to console this value ie:

```
console.info(document.getElementById("myHeading").innerHTML);
```

However, in the console we will get an error:



```
✖ ▶ Uncaught TypeError: Cannot read property 'innerHTML' of null  
at (index):15
```

This is because we have tried to reference a DOM element before the browser has had chance to load the DOM. There are two solutions here:

1. Add an event that will trigger the code when the DOM has loaded.
2. Move the `<script>` to the bottom of the document before the closing `</body>` as the DOM will have loaded at that point.

Both are perfectly valid. Let's see how we could accomplish this with the first option.

Leave the `<script>` in the `<head>` of the document but place all the code inside of a function as follows:

```
<script>  
function init(){  
    console.info(document.getElementById("myHeading").innerHTML);  
}  
</script>
```

We now have a function called `init()`. Next we'll create an event to call the `init()` function when the document has loaded. Earlier, it was suggested that inline Javascript is not recommended. However, this is one use-case that is commonly used by developers.

Amend the `<body>` tag as follows:

```
<body onload="init()">
```

Save and test your file. You should now see the console outputting the contents of the `<h1>` tag.

Another solution for this issue is to place the code at the bottom of the file. With this second approach there is no need to have the `onload` event attached to the `<body>` or to wrap your code inside of a function.

Remove the `onLoad` event from the `<body>` tag and move the `<script>` tag to before the closing `</body>` tag.

You should now also remove the `init()` function so the code appears as:

```
console.info(document.getElementById("myHeading").innerHTML);
```

This could also be amended to use an immediately invoked function expression as follows:

```
(function(){
    console.info(document.getElementById("myHeading").innerHTML);
})();
```

Finally, it is common to store Javascript in an external for re-use. Create a *js* folder in your application structure and create a file called *main.js*.

Move the Javascript from inside the `<script>` into this external file. Replace the `<script>` tag in *index.html* with a reference to the external file.

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

CHANGING THE DOM

To change the value inside of the `<h1>` identified above, we can use `innerHTML` as this is a property that can both be read and set. Change your by removing the console call and replacing it with:

```
document.getElementById('myHeading').innerHTML = "Your Name";
```

Save and test your file and you should see that 'Your Name' appear in the DOM.

As well as changing the content of a HTML element we can change its attributes. This done with a method of the document object called `setAttribute()`. The `setAttribute()` method takes two parameters, firstly the attribute to add and secondly its value.

We would like to add the class 'currentPage' to the first `` in the menu. This class is already in the CSS for this file and will simply underline the targeted element. As such the `setAttribute()` called would be:

```
setAttribute('class', 'currentPage');
```

As this element does not have an ID we need an alternative approach to select it. To do this we could use the method `document.querySelector()`. This takes as parameters a CSS selector and will return the first match found. As the `` is the first one in the list we could use a CSS selector of:

```
nav ul li
```

As such add the following to the Javascript code:

```
document.querySelector('nav ul li').setAttribute('class', 'currentPage');
```

Save and test your file. The 'Home' link should now appear underlined as a result of the class.

To show the power of this we'll change the link on the home button such that it points to <http://www.google.co.uk>.

To do this add:

```
document.querySelector('nav ul li a').setAttribute('href',  
'http://www.google.co.uk');
```

Test the page. You may want to remove this after proving it is possible!

Note: This isn't really something you'd be likely to do but does illustrate why developers need to be wary of a hacker attack known as Cross Site Scripting (XSS) where a hacker will attempt to run their Javascript on your page.

EVENTS

Events are a crucial part of Javascript development, so much so that Javascript is often described as an 'event driven' scripting language. There are a huge amount of events that Javascript can react to including click, onload, mousemove, mouseover, submit and even gestures like touchstart.

Events can be added a number of different ways.

Inline Events

Inline events are where the event is placed directly in the HTML code eg:

```
<p onClick="console.info('hi')">Test Event</p>
```

We have already seen a use of this with the `<body onload="init()">`. However, apart from this common use case inline events are discouraged as they make code hard to manage.

Note: With inline events the event itself is case insensitive, so all of these are equally valid: `onclick`, `onClick`, `ONclick`.

DOM Events

DOM events reference DOM elements and set an event property to point at a function. This function contains the code triggered as a result of the event and is known as an event handler. For example, in the code below the property `onclick` is used to reference an anonymous function that acts as the event handler.

```
document.getElementById('myTestEvent').onclick = function(){  
    console.info('hi');  
}
```

With DOM event the event property is case sensible and is in lowercase (not camelCase), so `onClick` should be invalid.

The example above uses an anonymous (ie unnamed function). Equally functions can be referenced by name eg:

```
document.getElementById('myTestEvent').onclick = myFunction;
function myFunction() {
    console.info('hi');
}
```

Notice that the `onclick` property points at the named function, it does not call it. As such there is no need for the parenthesis `()` after the function name.

USING THE ADDEVENTLISTENER() METHOD

The recommended way to work with events is the `addEventListener()` method. It is more flexible than the previous two approaches and is part of the DOM Events specification.

At its most basic the `addEventListener()` method takes two parameters. These are an event type and then an event handler.

The event can be any number of different events such as 'click', 'mouseover' or 'submit'

The event handler is a function, either named or anonymous, that is called when the event is triggered. An example of an anonymous function as the event handler would be:

```
document.getElementById('myTestEvent').addEventListener('click',
function() {
    console.info('hi');
})
```

It could also reference a named function eg:

```
document.getElementById('myTestEvent').addEventListener('click',
myFunction);
function myFunction() {
    console.info('hi');
}
```

As with the DOM event the `addEventListener` registers the function to call, thus the lack of parenthesis.

THE EVENT OBJECT

One useful feature of both DOM events and `addEventListener` is the ability to pass the event object itself to the event handler function. This is done by default but to expose this object we need to add a parameter to the event handler function. You can call this anything you like (within Javascript naming rules). Here we are using `ev` as it is nice short way to refer to an event.

```
document.getElementById('myTestEvent').addEventListener('click',
function(ev) {
    console.dir(ev);
})
```

Open the console and you will see that you have a `MouseEvent`. There are a lot of interesting properties and methods such `target` which indicates the target of the event.

CHANGING THE BACKGROUND COLOUR OF THE PAGE

In the *index.html* there are a number of `` tags designed to allow users to change the background colour of the file.

Create an event for each of the red, blue and green options to add the appropriate CSS class to the body tag. To do so you'll need to use `querySelector()` and `setAttribute()`.

Create an event for the reset option to set the background colour back to white. To do so use the `removeAttribute()` method.

ARRAYS IN JAVASCRIPT

Arrays are created in Javascript with either:

```
var myArray = new Array();
```

Or more efficiently with just:

```
var myArray = [];
```

In either case values can be added as a comma separated list ie:

```
var myArray = new Array('cat', 'dog', 'mouse');
```

Or

```
var myArray = ['cat', 'dog', 'mouse'];
```

As arrays represent more complex variable add them to the console with `console.dir()`.

```
console.dir(myArray);
```

Array values are accessed via their index number eg:

```
myArray[0]; // cat
myArray[2]; // mouse
```

Indexing starts from zero.

Loops are commonly used to extract values from an array:

```
for (var i=0; i<3; i++){
    console.info(myArray[i]);
}
```

The array property `length` can be used to find the number of values in an array.

CREATING AN ARRAY OF IMAGES

In the file *index.html* on line 23 is an `` with an ID of `myImages`. We'll create an array to change the image displayed.

In the *js/main.js* file create an array as follows:

```
var imageAr = ['images/view1.jpg', 'images/view2.jpg', 'images/view3.jpg',
'images/view4.jpg', 'images/view5.jpg', 'images/view6.jpg'];
```

Any of these images can be assigned to the image `src` attribute with for example:

```
document.getElementById('myImages').setAttribute('src', imageAr[2]);
```

To have the images rotate we could use the Javascript `setInterval()` method. This will call a given method/function at a set interval defined in milliseconds.

To test this, create a function as follows:

```
function chgImage(){
    console.info('Called');
}
```

and then reference the function with a `setInterval()`:

```
setInterval(chgImage, 2000);
```

This will change the value every 2 seconds. We want to use this to change the image source so need a counter to help. Declare a counter variable and increment it in the `chgImage()` function. Use conditional logic and the `length` property to loop the selected image back to the beginning of the array if the last image is been displayed.

WORKING WITH FORMS

HTML forms are often associated with Javascript, as it can be used to retrieve form values and to perform tasks such as form validation.

Inside the file *exchangeRate.html* there is a basic HTML form. We'll use this form to build a simple currency convertor. Create a second external Javascript file in *js/exchange.js*.

Add `<script>` tag to the bottom of the file just before the closing `</body>` to load this external file as follows:

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

Inside `js/exchange.js` add the following immediately invoked function expression:

```
(function(){  
    // logic here  
})();
```

For this application we'll need five variables. There are three values needed for the calculation. That is the value in pounds to be converted, the exchange rate and the value in euros. Declare these inside of the immediately invoked function:

```
var pricePounds;  
var exchangeRate;  
var priceEuros;
```

Then we could also use a variable representing the HTML form and a HTML element used to feedback to the user.

Notice that the `<form>` has an ID. As such we can reference it with `document.getElementById`. We also have a `<p>` of ID `msg` that we'll use to feedback to the user. This is currently set to `display:none` in the CSS. Add two new variables as follows:

```
var convertForm = document.getElementById('myForm');  
var msg = document.getElementById('msg');
```

HANDLING FORM SUBMISSION

When the submit button is pressed the HTML form would normally be submitted to the action of the form. In this application we don't want that to happen as it would by default cause a page refresh. We would like a form submission to perform the currency conversion.

With Javascript we can listen out for events that occur and capture them. In this example we listen out for the form submission and capture it to do our currency conversion. We'll also stop the form from submitting which is its default behaviour.

To capture the submit event we'll use `addEventListener()`. This will also allow us to expose the event object.

Add the following:

```
convertForm.addEventListener('submit', function(ev) {
    ev.preventDefault();
})
```

The above adds an event listener to the form and listens for when it is submitted. When it is submitted the event triggers the anonymous function.

As we have passed the event object to the function and exposed it as `ev` we can call a method of the event object `preventDefault()` that will stop the form submitting in the normal way. That way we can perform our calculation logic and display the result to the user without a page refresh.

RETRIEVING THE FORM VALUES

To retrieve the values from the form we can use the property `value`.

A value in pounds is entered by the user into this field:

```
<input type="text" id="pounds">
```

We can therefore retrieve that value with:

```
document.getElementById('pounds').value;
```

The `value` property, like `innerHTML`, can be used to get and set the value.

With HTML forms all values are returned as strings not numbers, so even if the user entered 12 it is a string value of '12'.

To convert the value to a number we can use the method `parseFloat()`. This will take a value and convert it into a floating point (decimal). If it cannot convert the value then it will return `NaN` (not a number).

Note: As well as `parseFloat()` Javascript has other methods such as `parseInt()` and `parseFloat()` to help with data type conversion.

Inside of the event handler we could now add:

```
priceEuros = 0;
exchangeRate = 0.87;
pricePounds = parseFloat(document.getElementById('pounds').value);
```

If the value entered by the user is not a number, we need to provide a suitable message. The `isNaN()` method can be used to see if a value is a number.

```
if(isNaN(pricePounds)){
    msg.style.display = "block";
    msg.innerHTML = "You must enter a number";
    msg.setAttribute('class', 'error');
}
```

If the variable `pricePounds` is not a number (`isNaN`) then the HTML of ID `msg` is displayed. This is done using `style` object of the DOM element and setting its `display` property to `block`.

This adds an inline CSS style. When initially styling a file, inline styles are usually frowned upon but as this style is added via Javascript as part of an inaction with the user is it a perfectly valid approach. We also add a suitable error message via `innerHTML` and use `setAttribute()` to add the class of `error` which is already defined in the CSS.

If the number is valid add an else block to the above if condition. Inside this else block we'll check to see if the number entered is zero.

```
if(pricePounds === 0){
    msg.style.display = "block";
    msg.innerHTML = "You must enter more than zero."
    msg.setAttribute('class', 'error');
}else{
    priceEuros = pricePounds * exchangeRate;
    msg.style.display = "block";
    priceEuros = priceEuros.toFixed(2);
    msg.innerHTML = "You will get &euro;" + priceEuros;
    msg.removeAttribute('class');
}
```

If the value is zero then a suitable error message is displayed. If not then the calculation is performed and the result displayed back to the user. Note the user of the HTML entity to ensure consist display of the euro sign.

RESETING THE INPUT FIELD WITH THE 'THIS' KEYWORD

Finally, we'll reset the input field for the pounds value when the user places their focus in it.

As a new event add:

```
document.getElementById('pounds').addEventListener('focus', function(){
    this.value = "";
    msg.innerHTML = "";
    msg.removeAttribute('class');
})
```

This uses a `focus` event. A focus event is triggered when a form field receives the focus of the cursor. Once triggered we clear the error message and its error message status.

The `this.value` uses the `this` keyword which refers to the current context. Here this refers to the DOM element that has the event handler attached to it ie:

```
document.getElementById('pounds')
```

As such here, `this.value` refers to the value of input field.

USING THE EVENT OBJECT

Alternatively, the above could be achieved by exposing the event object in the event handler function as follows:

```
document.getElementById('pounds').addEventListener('focus', function(ev){
    ev.target.value = "";
    msg.innerHTML = "";
    msg.removeAttribute('class');
})
```

Here the event is exposed as `ev` and then the value extracted via `ev.target.value`. The target property of the event object refers to the target of the event.

ADVANCED OPTION: USING AN ARROW FUNCTION

With ES6, functions can be written using syntax known as an arrow function `=>`.

```
document.getElementById('pounds').addEventListener('focus', (ev)=>{
    ev.target.value = "";
    msg.innerHTML = "";
    msg.removeAttribute('class');
})
```

Warning: With arrow function the behaviour of the `this` keyword changes. An arrow function does not have its own `this` context and as such will look for `this` in higher scopes. As such in the above example we would need to use the event object.