Edinburgh Napier University

SET08101 Web Tech

Lab 7 - JavaScript: Core Language

Dr Simon Wells

1 Aims

At the end of the practical portion of this topic you will:

- Use inline JavaScript
- Use <script> tags
- Use external JavaScript files

NOTICE: Just as with our other core web languages, HTML & CSS, there is a lot of useful material online. In addition to reading the relevant chapters of the module texts, you should also avail yourself of the following which document JavaScript:

- https://developer.mozilla.org/bm/docs/Web/JavaScript
- https://www.w3schools.com/jsref/default.asp

2 Activities

We're going to tackle JavaScript in two parts: first as "core JavaScript", dealing with JavaScript as a programming language, and then as "client-side JavaScript", handling interaction with HTML pages, CSS, the user-agent (browser) environment, and wider Web (via APIs).

This core part will deal with the basic relationship between HTML and JavaScript, insofar as we need HTML to host our JavaScript to get it into the browser. Once our JavaScript is in the browser, we can treat it as code and program things to our hearts content. We'll then consider JavaScript as a programming language, divorced from its web and browser context. Then, next week, once we have some idea of JavaScript as a standalone language, we'll start to use it to manipulate our web pages and wider web-environment.

We'll start by investigating three ways to integrate our JavaScript with a website. We could alternatively, just type JavaScript into the terminal of our browser and have it executed interactively (which is a really valuable way to try out bits of code), but the main way that we'll write JavaScript is in the context of a web page. This is because of the way that HTTP works. Our browser makes a request for a hypertext document, i.e. HTML, then this contains links to all of the other files and documents that it is related to. Our browser retrieves the files pointed to by these links as required in order to build your webpage. So to get our JavaScript to our browser we need, at a minimum, some HTML document that can serve to pull our JavaScript to the browser. Our choices for integrating JavaScript with our sites are then to integrate our JavaScript directly amongst our HTML (which is less than ideal but useful for trying out ideas), embedding all of the JavaScript code in a script block (for example in the head of the document), or else to have a separate, external file containing only JavaScript, that is linked to from out HTML. Each approach has advantages and disadvantages, reminiscent of the similar choices when considering the relationship of CSS to HTML and the use of inline, block, or external CSS.

3 Inline JavaScript

In rare circumstances we might want to add some JavaScript direct to an HTML element like so:

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>SET08101 - Inline JavaScript</title>
5     </head>
6     <body>
7         <p><a href="#" onClick="alert('Hello World');">Click Me</a></p>
8     </body>
9 </html>
```

Congratulations, you just wrote your first JavaScript[1]. We won't often do this, just like we won't often attach CSS styles direct to an element within HTML. However, it does keep things incredibly simple.

Load up the HTML file that you just wrote and try it out. Now let's modify that a little:

```
<!DOCTYPE html>
<html>
    <head>
        <title>SET08101 - Inline JavaScript</title>
    </head>
    <body>
        <p><a href="#" onClick="console.log('Hello World')">Click Me</a></p>
    </body>
</html>
```

This time you'll notice that there was no longer any alert box displayed when we clicked the link. Instead, to see where our output went, we need to find our *console*. The console is provided by your browser as part of the Developer Tools[2]. The console is a bit like a command line built into your web browser that will display the output of calls to the console.log() function. This is a really useful function because we'll use it a lot to get output from our code, for example, when we want to test that the value actually stored in a variable actually matches what we think it is[3].

Something else to note in our two examples so far is the use of *quoting;* that is the use of the ' and " marks around things. Note that double quotes were used to surround everything associated with onClick= and that within the onClick single quotes were used around the string 'hello world'. There is no rule that says we have to use single quotes inside double quotes, we could have had things the other way around, we just have to be consistent, i.e. ensure that pairs of single quotes match each other and similar for double quotes. We also need to remember that we can't have a pair of double quotes within another set of double quotes, as the opening inner double quote will close the outer double quote and the rest of the line will be, at least, a syntax error. Try it out a bit so you get the idea, e.g.

- onClick='console.log("Hello World")'
- onClick='console.log('Hello World')'
- onClick="console.log("Hello World")"

For each one, see what happens in the console. Get used to checking the output in the console. It's not just for the expected output from calls to console.log() but also will tell you when there are errors.

There are other HTML events, besides onClick, that we can respond to, for example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>SET08101 - Inline JavaScript</title>
    </head>
    <body>
        <p><a href="#" onMouseOver="alert('Hello');">Hover over me</a></p>
    </body>
</html>
```

Note the difference between user interaction, i.e. when the event triggers based on your moving your pointer.

[1]Unless you've written some JavaScript before, in which case, well done, you just wrote some more JavaScript.
[2]If you can remember right back to the week one, this was one of the first things that we got to grips with.
[3]There are alternatives to outputting the values of variables when we are developing, but this is a useful place to start.

```
<!DOCTYPE html>
<html>
    <head>
        <title>SET08101 - Inline JavaScript</title>
    </head>
    <body>
        <p><a href="#" onMouseOut="alert('Hello');">Hover over me</a></p>
    </body>
</html>
```

Different events give use lots of control over how the user can interact with our pages and the elements that make up our pages. There are many other HTML DOM events that your code can respond to. Explore them here: https://www.w3schools.com/jsref/dom_obj_event.asp.

3.1 Using Script Blocks

As for CSS, using inline JavaScript feels a bit hacky, good enough for trying something out but not so good if we want to build a larger or more manageable site. Remember, the more mixed up and disorganised things are, the less easily they can be found when needed, or altered with good knowledge of any possible side effects.

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">
        function helloFunction() { alert('Hello Napier'); }
        </script>
        <title>SET08101 - External JavaScript</title>
    </head>
    <body>
        <p><a href="#" onClick="helloFunction();">Click Me</a></p>
    </body>
</html>
```

3.2 External JavaScript Files

Just as with CSS, we can also put all of our JavaScript into external .js files which can be linked to any html that needs access to the JavaScript code contained therein. This way we can update our JavaScript in just one place and have our changes available everywhere they are needed. An additional advantage is that external JavaScript files can be cached, this means it is basically saved for reuse. If two pages use the same JavaScript external file then instead of loading it twice, as would happen if the JavaScript was between <script> tags in the respective HTML files, instead it is downloaded for the first page that your user navigates to then is saved in the browser and reused, but not re-downloaded, when referenced from the other file. This can lead to large saving in network bandwidth but also better performance of your sites as loading from local memory is nearly always much faster than reloading across the network.

Let's look at an example. Create a text file called external.html and place the following code into it:

```
<!DOCTYPE html>
<html>
    <head>
        <script src="hello.js"></script>
        <title>SET08101 - External JavaScript</title>
    </head>
    <body>
        <p><a href="#" onClick="helloFunction();">Click Me</a></p>
    </body>
</html>
```

Now create a second text file in the same folder as external.html called hello.js and place the following code in it:

```
function helloFunction(){
    alert('Hello Napier');
}
```

Note that we could have named our JS file however we like, it just needs to match the name we put in our HTML so that the correct file can be found and loaded by the browser. As with our CSS files, we can also place our JavaScript files into their own folders but we must then give the *path* to our JavaScript within the script tag. This flexibility gives you options for keeping your "codebase" nicely organised so that you can easily find things later. How exactly you organise things is up to you, but a good default is to group like files together in appropriately named sub-folders, e.g. "css" for your style sheets and "js" for your JavaScript.

Note that you will see <script> tags placed in various places within an HTML file, for example, within the <head> or within the <body> but either is valid. In the past, it made a difference to the end user because it could affect their experience. Browsers would tend to make pages available to the user as they loaded, progressively adding functionality until everything was loaded. This was noticeable in the past when the web was very slow but is less prevalent nowadays. The issue was related to the idea that if a script was loaded in the head then it was available sooner, but you couldn't guarantee that everything the script depended upon was loaded, which lead to the argument for adding script tags at the end of the HTML, but which meant that functionality in the script was only available once everything was loaded (which could be a while if there were lots of other files, e.g. images, that needed to be loaded first).

From this point on you should decide whether to use script blocks or external JavaScript files. It can be easier when trying things out to do everything on the same page, but as soon as you start working on a larger project it can be worth extracting everything out into its own file, if only to enable you to reuse your JavaScript across multiple pages.

4 Core JavaScript: JavaScript as a programming language

We've considered just enough JavaScript and HTML to get a page loaded that doesn't do very much. Next week we'll consider how we can use JavaScript to interact with our pages, amongst other things, but that presupposes that we can program in JavaScript. This isn't a first programming course, you should all have some experience of programming in some form of language, and the chances are that that language is a C-style language. This just means that the language has a C-style of syntax, the benefit of which is that, although the details will escape you at this point, you should be able to "read" the code and infer some of what the code is trying to do. Things like variable assignment, expressions, code order and flow (branching and iteration), and code organisation (functions and objects) should all be familiar, if a little different to what you are used to. Remember that things are different, and even things that look the same might behave in a slightly different way, but that you can get some idea of what is happening by using your existing language knowledge as a starting place to develop your new knowledge of JavaScript.

A good place to start is with the W3Schools JavaScript exercises:

- JavsScript Exercises: https://www.w3schools.com/js/

These will introduce you to all of the basics of the language so that you are aware of the available features and functionality. Don't worry about learning everything right now, you will become familiar with things that you use frequently as you progress and write more programs, but for now, just knowing that features exist, even if you need to look up the details, is enough.

We're going to become familiar with JavaScript by writing a series of small programs to solve some simple problems. Use the "external file" framework from above as a starting place, and type your JavaScript into the .js file. After you've made changes to your file you may need to refresh the page. Because of caching, where the browser keeps a copy of your files locally, it can be helpful to use a "private" or "incognito" page during development. Any output from your code should be written to the console using console.log() expressions. We'll worry later about displaying results in our HTML pages and otherwise manipulating browser content.

For each of the following you should consider what you are being asked to do, perhaps think about what a solution in a different language might look like if it helps. Once you have an idea of the shape of a solution, e.g. the arguments being passed into your function, and the result being returned, you should be able to create a basic JavaScript function outline. From there you should be able to work out what the expressions might be to compute the result. Note that each thing that is introduced may require you to investigate the JavaScript language documentation, for example, to see how arrays are handled, or declared, to see how functions are handled, and to see how expressions are constructed.

Write a JavaScript function to identify the type of a given angle that's passed in as an argument. Rember that there are different types of angle, for example, the acute angle (An angle between 0 and 90 degrees), the right angle (An 90 degree angle), the obtuse angle (An angle between 90 and 180 degrees), and the straight angle (A 180 degree angle).

Write a JavaScript function to replace each character of a string, supplied as an argument to the function, by the next one in the English alphabet, returning the resulting manipulated string.

Given an array of strings, write a JavaScript function to find the longest string in the supplied array.

4.1 Finally

A good way to practice a new language or to improve your existing abilities, not just in JavaScript, but in any language you might be trying to learn, is to try to solve a set of problems using the language. I often use Project Euler when starting out with a new language but there are also many others:

1. Project Euler: https://projecteuler.net/
2. Stack Exchange Code Golf: http://codegolf.stackexchange.com/
3. Code kata: http://codekata.com/
4. Reddit Daily Programmer: https://www.reddit.com/r/dailyprogrammer
5. Programming Praxis: http://programmingpraxis.com/
6. Rosetta Code: http://rosettacode.org/wiki/Main_Page
7. International Collegiate Programming Contest Problems Index: http://acm.hit.edu.cn/judge/ProblemIndex.php
8. Algorithmist: http://www.algorithmist.com/index.php/Main_Page

Another trick I have is to have a short list of small coding projects that Dr. Simon Wells redoes whenever he learns a new language. His personal favourites are to write programs that deal with the following topics (but you should put together your own list if his interests don't match yours):

1. Writing 1D and 2D Cellular Automata
2. Conway's Game of Life
3. Simple text based games, e.g. story-telling or adventure games
4. Quines
5. Simple codes & ciphers

By repeating exercises that you're already familiar with in a new language you can quickly get a feel for a new language does things differently or offers features that make a given problem easier or more difficult to tackle.

This isn't the end of our JavaScript journey, but just the beginning, we're going to spend time getting used to client-side JavaScript, then we'll look at server-side JavaScript later on.