COMP519 Practical 7
JavaScript (2)

Introduction

- This worksheet contains exercises that are intended to familiarise you with JavaScript Programming. While you work through the tasks below compare your results with those of your fellow students and ask for help and comments if required.
- This document can be found at

  https://cgi.csc.liv.ac.uk/~ullrich/COMP519/notes/practical07.pdf

  and you might proceed more quickly if you cut-and-paste code from the PDF file. Note that a cut-and-paste operation may introduce extra spaces into your code. It is important that those are removed and that your code exactly matches that shown in this worksheet.
- The exercises and instructions in this worksheet assume that you use the Department's Linux systems to experiment with JavaScript.

  If you want to use the Department's Windows systems instead, then you can do so.
- To keep things simple, we will just use a text editor, a terminal, and a web browser. You can use whatever text editor and web browser you are most familiar or comfortable with.
- If you do not manage to get through all the exercises during this practical session, please complete them in your own time before the next practical takes place.

Exercises

1. Let us continue our consideration of mathematical operators in JavaScript. At the same time we want to get a better seperation of JavaScript code from HTML markup.

   a. Start a new file `jsDemo07A.html` in your `public_html` directory with the following content:

```
<!DOCTYPE html>
<html lang="en-GB">
  <head>
    <title>JavaScript 07A</title>
  </head>
  <body>
    <p>Our third JavaScript script</p>
    <script src="jsDemo07A.js"></script>
  </body>
</html>
```

   Then create a separate file `jsDemo07A.js` in your `public_html` directory with the following content:

```
for (var ii = 4, jj = 3; jj >= 0; ii++, jj--) {
   document.writeln(ii + " * " + jj + " = " + ii*jj + "<br>")
   document.writeln(ii + " / " + jj + " = " + ii/jj + "<br>")
   document.writeln("log(" + jj + ") = " +
                    Math.log(jj) + "<br>")
```

```
   document.writeln("sqrt(" + (jj-1) + ") = " +
                     Math.sqrt(jj-1) + "<br><br>")
}
```

b. Save the files and make sure that their access rights are set correctly, that is, the access rights of `jsDemo07A.html` and `jsDemo07A.js` should be set as in Practical 6.

c. Open `jsDemo07A.html` in a web browser and inspect the output that our JavaScript script has produced. Make sure that you understand how results like `NaN`, `Infinity`, and `-Infinity` come about.

d. The first four lines of output produced by our JavaScript script are:

```
4 * 3 = 12
4 / 3 = 1.3333333333333333
log(3) = 1.0986122886681098
sqrt(2) = 1.4142135623730951
```

As you can see the results of the mathematical operations are displayed with quite a high number of decimal places. Modify the JavaScript code so that at most three decimal places are displayed, that is, the output should be

```
4 * 3 = 12.000
4 / 3 = 1.333
log(3) = 1.099
sqrt(2) = 1.414
```

Hint: Search on-line for a JavaScript function that limits the number of decimal places in the conversion of a number to a string.

e. Extend `jsDemo07A.js` with the following code:

```
x = 1.275
y = 1.27499999999999991118
document.writeln(x + " and " + y + " are " +
                 ((x == y) ? "equal" : "not equal") + "<br>")
```

f. Save the file and refresh the page `jsDemo07A.html` in your web browser. The last line of the output should now be:

```
1.275 and 1.275 are equal
```

While this statement in itself it is true, something strange seems to have happened to the value of the variable `y`. What is going on?

2. Your JavaScript code is accessible and readable by everyone who can access your web page. If you want to make it a bit more difficult for others to read, copy, and/or modify your code, then you should obscure it, for example, using `uglifyjs`.

a. In a terminal, go to the directory in which the files `jsDemo07A.js` and `jsDemo07A.html` has been stored and execute the commands

```
mv jsDemo07A.js jsDemo07A.pretty.js
uglifyjs jsDemo07A.pretty.js -m -c > jsDemo07A.js
```

(The option "`-m`" is for `mangle` and the option "`-c`" is for `compress`; to see a full list of options, use `uglifyjs -h`.)

b. Check that `jsDemo07A.html` still produces the same output as before. Compare the files `jsDemo07A.js` and `jsDemo07A.pretty.js` to see how they differ.

When you use this method for the COMP519 JavaScript assignment, then make sure that you submit both readable/pretty code as well as the unreadable/uglified code via the departmental submission system but place only the unreadable/uglified code in your `public_html` directory.

3. Hopefully, so far all your JavaScript code worked without problem. Once you write your own code, this is not likely to always be the case. If your code contains syntax errors, you have already seen that the console can give you an indication where those errors are. But logical errors are harder to find. Your code contains *logical errors* if the code does not produce any syntax or runtime errors but produces an incorrect result or incorrect behaviour. Here we look at two ways that you can use to pinpoint logical errors in the code.

   a. Start a new file `jsDemo07B.html` in your `public_html` directory with the following content:

```
<!DOCTYPE html>
<html lang="en-GB">
  <head>
    <title>JavaScript 07B</title>
  </head>
  <body>
    <script>
    for (i = 1; i<6; i++) {
      mode = Math.floor((Math.random()*4)+1)
      /* Add debugging code here */
      switch (mode) {
        case 1:  randvar = Math.round(Math.random()*100);
                 break
        case 2:  randvar = String(Math.random())
                 break
        case 3:  randvar = Math.random()/(Math.random()+1)
                 break
        default: randvar = Boolean(Math.random()+0.5)
                 break
      }
      document.writeln("Random value: (" + typeof(randvar) + ")"
                       + randvar + "<br />\n")
      }
    </script>
  </body>
</html>
```

   b. Save the file and make sure that their access rights are set correctly.

   c. Open `jsDemo07B.html` in a web browser and inspect the output that our JavaScript script has produced. The output might be something like this:

```
Random value: (number)0.4064974478835452
Random value: (string)0.0037800505449638866
Random value: (boolean)true
Random value: (boolean)true
Random value: (string)0.4909139084492823
```

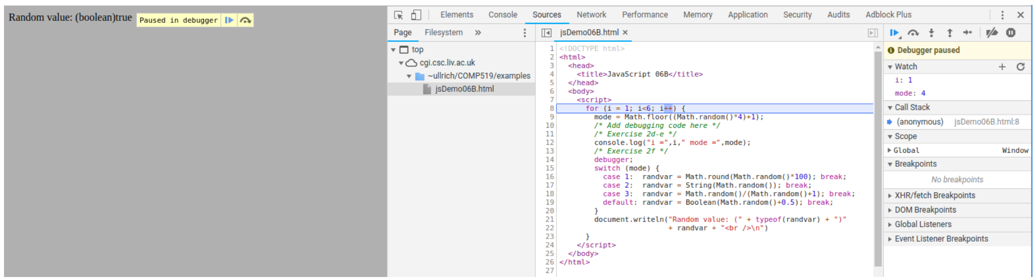

Figure 1: JavaScript Debugger in Google Chrome

d. Suppose that you are not sure what branch of the switch-statement is executed in each iteration of the for-loop and you would like to know what the values of the variables `i` and `mode` in each iteration of the for-loop are to figure that out. To get this information you can add the following *debugging code* at the point indicated in the code.

```
console.log("i =",i," mode =",mode)
```

e. Save the file, reload `jsDemo07B.html` in your web browser, then have a look at the JavaScript console of the web browser. It should contain output like

```
i = 1  mode = 1            jsDemo07B.html:11:9
i = 2  mode = 2            jsDemo07B.html:11:9
i = 3  mode = 4            jsDemo07B.html:11:9
i = 4  mode = 4            jsDemo07B.html:11:9
i = 5  mode = 2            jsDemo07B.html:11:9
```

So, `console.log` wrote its output to the console, not the HTML document, and you can now see what the values of `i` and `mode` are. The number 11 after `jsDemo07B.html:` indicates that the output was produced by code in line 11 of your script.

f. An alternative or additional facility that you can use to find logical errors in a JavaScript program is the *debugger* that browsers like Google Chrome and Mozile Firefox provide. In the following we focus on Google Chrome. Replace the line introduced in 3d by the following

```
debugger
```

g. Save the file, make sure that the developer tools are already open in the browser, then reload the file. In the developer tools you should now see the HTML markup and JavaScript code of the page and the line containing `debugger` is highlighted (Figure 1). [1]. In the rightmost column, just above the line 'Debugger paused' you see a number of icons that control the debugger. Below that are various panels with additional information and functionality.

h. We will use one these functions, namely, *Watch*, that allows us to keep an eye on the value of a variable or expression. To add a variable that you want to watch, click on the plus symbol to the right of `Watch` then type in the name of the variable into the field that appears. Use this to watch the variables `i` and `mode`.

i. Now proceed with the execution of the code. Use the ▶ button to 'resume execution' of the code. The execution will again stop when it encounters the `debugger` statement again which is at the next iteration of the for-loop. Observe how output starts to appear in the browser window and how the values of the watched variables change.

j. The debugger has a lot of other useful features, for a more comprehensive tutorial on its use see [1].

4. The following exercise concerns *types*, *equality* and *comparisons* in JavaScript.

   a. Start a new file `jsDemo07C.html` in your `public_html` directory with the following content:

```
<!DOCTYPE html>
<html lang="en-GB">
  <head>
    <title>JavaScript 07C</title>
  </head>
  <body>
    <script>
    Array.prototype.toString = function () {
      var arrStr = "["
      for (var i=0; i<this.length; i++) {
         arrStr += this[i]
         if (i<this.length-1) arrStr += ", "
      }
      return (arrStr += "]")
    }

    var values = [0, 123, 1.23e2, "12.3e1", true, false, NaN,
                  Infinity, undefined, null, [0]]
    for (var i=0; i<values.length; i++) {
      document.writeln(i+") The type of "+values[i]+
                       " is: "+typeof(values[i])+"<br>")
    }
    document.writeln(i+") The type of "+values+" is: "+
                     typeof(values)+"<br>")
    </script>
    <noscript>JavaScript not supported or disabled</noscript>
  </body>
</html>
```

   b. Save the file and make sure that its access rights are set correctly, that is, the access rights for `jsDemo07C.html` should be set as in Practical 6.

   c. Open `jsDemo07C.html` in a web browser and inspect the output that our JavaScript script has produced. Make sure that you understand why each array element is printed in the way it is and why the types of each array element are what they are.

   d. Extend the script with the following code:

```
document.writeln("Elements of values are: " + values.join()+"<br>")
```

   The output produced by this additional code should be:

```
Elements of values are: 0,123,123,12.3e1,true,false,NaN,Infinity,,,[0]
```

   Where you would expect `'undefined'` and `'null'` to appear, there seems to be an empty string. This is although `join` is meant to concatenate the string representations of the elements of an array and the string representations of `undefined` and `null` are `'undefined'` and `'null'`, respectively.
   To gain a better understanding of what `join` really does, refer to [2, Section 22.1.3.13].

   e. Extend the script with code that compares every element of `values` with every other element of `values` and itself using each of the four operators ==, ===, >, and <. The code should print out the elements it compares, their types and the result of the comparison.

The first nine lines of the output produced by your code should look as follows:

```
(number)0 == (number)0: true
(number)0 === (number)0: true
(number)0 < (number)0: false
(number)0 > (number)0: false

(number)0 == (number)123: false
(number)0 === (number)123: false
(number)0 < (number)123: true
(number)0 > (number)123: false
```

f. Change the code developed in the previous step so that the results of the comparison are presented as an HTML table. For example, the nine lines above should produce two rows in the table:

(number)0 == (number)0: true	(number)0 === (number)0: true	(number)0 < (number)0: false	(number)0 > (number)0: false
(number)0 == (number)123: false	(number)0 === (number)123: false	(number)0 < (number)123: true	(number)0 > (number)123: false

References

[1] Kayce Basques. *Tools for Web Developers: Get Started with Debugging JavaScript in Chrome DevTools*. Google. 22 August 2018. URL: https://developers.google.com/web/tools/chrome-devtools/javascript/ (accessed 10 October 2018).

[2] Ecma International. *ECMAScript 2018 Language Specification*. June 2018. URL: http://www.ecma-international.org/ecma-262/9.0/ (accessed 10 October 2018).