Lab 05 – JavaScript Basics

Aims:

- Understand the structure of a JavaScript file and the purpose of each part.
- Link an external JavaScript file to an HTML page.
- Create function that responds to browser events.
- Create an interactive Web page with JavaScript prompts and message alerts.
- Use JavaScript to read user input and write back to an HTML page.
- Debug JavaScript using both the Firefox Error Console and Firebug.

Show Task 1 to your tutor so your work can be marked off. Task 2 & 3 are optional.

Task 1: Dynamic User Interaction using JavaScript

Dynamic **behaviour** can be added to presentation of a Web page using **CSS**, e.g. with **pseudo classes** like `hover`, or through **JavaScript**. Here we will use JavaScript.

Step 1: Create an HTML file with user interaction

First we will create a simple HTML page that can take some user input. In a new folder **`lab05`**, create a Web page called **`clickme.html`** with the appropriate **metatags**, **title**, **heading** and **paragraph** as follows:

Typically to display a button we would use a form with an `<input type="submit">`, but here we will use a `<button>` element instead, because we are not sending anything to the server.

Load the page into Firefox and validate the HTML before you proceed.

Step 2: Create a link to a JavaScript file in the HTML file

To provide separate behavior to a Web page using a JavaScript file, the Web page must have a reference to the JavaScript file (in a similar way it needs references to CSS files). Add the following line in the <head> section to create a link to the (as yet non-existent) JavaScript file called 'io.js' in the same folder as the HTML page:

```
<head>
        …
        <script src="io.js"></script>
</head>
```

NOTE: Unix/Linux file names are Case Sensitive. Make sure the reference in your HTML exactly matches the file name.

Step 3: Create an external JavaScript file

Create the JavaScript file **`io.js`** using NotePad++ or similar text editor

1. Create a JavaScript template

Create a copy of the following JavaScript code template replacing the text in *italics* as appropriate:

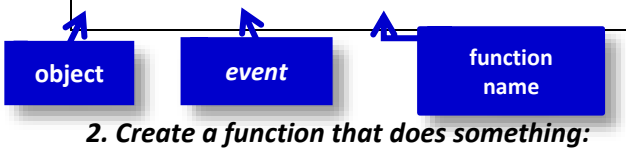
```
/**
* Author: Your name and student Id
* Target: What html file are reference by the JS file
* Purpose: This file is for …
* Created: ??
* Last updated: ??
* Credits: (Any guidance/help/code? Credit it here.)
*/

"use strict";     //prevents creation of global variables in functions

// this function is called when the browser window loads
// it will register functions that will respond to browser events
function init() {

}

window.onload = init;
```



2. Create a function that does something:

The pattern for a function definition is as follows:

```
/* Short comment on what the function does goes here*/
function myFunction(parameters) {
      // ...your JavaScript code to handle the event goes here
}
```

We will now use this pattern to display a prompt box. This box will take an input from the user then assign to a string. We will then display that string in an alert box.

Just above the `init()` function in your JavaScript file, create a function with no parameters called `promptName()` and insert the following lines between its braces:

```
var sName = prompt("Enter your name.\nThis prompt should show up when
          the\nClick Me button is clicked.", "Your name");
alert("Hi there " + sName +". Alert boxes are a quick way to check the
          state\n of your variables when you are developing code.");
```

What does this last parameter do?

Notice the use of the **`\n`** escape characters in the above strings to force the text onto a new line.

3. Register the function to respond to events in the browser:

The function you have written does not do anything yet because nothing is calling it. We will get our JavaScript to respond to an *onclick* event when the HTML button element with attribute `id="clickme"` is clicked.

When the HTML loads in the browser window, we need to register the elements on the HTML page that will generate events to which the JavaScript will then respond. There are many types of HTML *events* we can get JavaScript to respond to. Window object events like: *onload* or *onunload*; form object events: *onblur*, *onfocus*, *onchange*, *onsubmit*; mouse events like *onclick*, *ondblclick*, *ondrag* and so on. (Notice events do *not* use camelCase.)

We can register events inside the `init()` function. First we need to get a reference to the HTML object. One way to do this is to use the `getElementById()` method. A reference to the HTML object is then stored in a local JavaScript variable. The code pattern for this is:

```
var myVariable = document.getElementById("id_of_an_HTML_element");
```

We can then use this variable to register a 'listener' function that will respond to an action of a user on the HTML page:

```
myVariable.eventType = myFunction;
```

Using the above pattern, register a function to respond to a click event on the button by adding the following lines to the init() function you created in your template above:

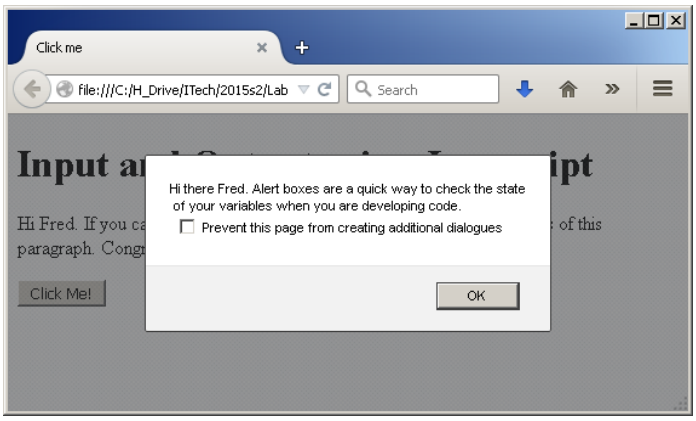
clickMe is a variable local to the `init` function. **Careful**: JavaScript is case sensitive!

```
var clickMe = document.getElementById("clickme");
clickMe.onclick = promptName;
```

Name of the function you wrote. Remember: no brackets

Step 4. Test your program, debug, deploy to mercury and retest.

Save your JavaScript file then refresh the HTML file. The output should appear something like this:



Upload your HTML and JavaScript files to a new **lab05** directory under the unit folder on the Mercury server **~/{your unit code}/www/htdocs**.

Open the page in Firefox, test that the JavaScript works.

Step 5. Writing output to the Web page.

We will now write some JavaScript code that will change the content of the paragraph element on our Web page. We will do this in a new function.

1. Create an id for an HTML element

First we need to create an id attribute for our paragraph element in the HTML. Modify your file **`clickme.html`** to add the attribute **`id="message"`** to the first <p> element. Resave your file and check that the HTML is valid.

2. Create a new function to write content into the HTML element

Create a new function called **`rewriteParagraph(userName)`**. Notice that unlike the function we created in Step 3.2, this function has a parameter called `username`.

```
function rewriteParagraph(userName){
        //1.  get a reference to the element with id "message" and assign it to a local variable
        // 2. write text to the html page using the innerHTML property
}
```

Function definition

Write the implementation of this function. In the first line, get a reference to the paragraph element with id attribute "message" and assign this reference to a local variable called **`message`** (as you did in Step 3.3).

The **`innerHTML`** property sets or returns the HTML content (inner HTML) of an element. In the second line of your function write text to the html page using the innerHTML property as follows:

```
message.innerHTML = "Hi " + userName + ". If you can see this you have
    successfully overwritten the contents of this paragraph. Congratulations!!";
```

3. Invoke the new function

Unlike the **`init()`** and **`promptName()`** functions above, we will not directly register a browser event to invoke our new function. Instead we will make a function call.

Within the **`promptName()`** function add the following function call at the end:

```
rewriteParagraph(sName);
```

Function call

This will pass a copy of the value in the **`sName`** variable that we got from the user to the **`rewriteParagraph`** function (where it become the local variable **`userName`**.

4. Test your program, debug, deploy to mercury and retest.
The result should now look something like this:

Step 6. Another way to write output to the Web page.

From the example file of the first JavaScript lecture (Lecture5), observe how the **`textContent`** property can be used inside a **`<span>`** element to write into a placeholder on the HTML page.

1. Modify your **`clickme.html`** file to include a paragraph <p> just above the button element. Inside the paragraph add a with an appropriately named id attribute.

2. Create a new function called **`writeNewMessage()`** that will display the message "You have now finished Task 1" into the span using the **`textContent`** property.

3. Modify your code so that when the user clicks on the <h1> element the **`writeNewMessage`** function is invoked.

4. Test your program, debug, deploy to mercury and retest.

The output should now look something like this:

Something optional to try:

We can use an alternative method to reference an element on an HTML page without having to create an id on the HTML page. The **`document.getElementsByTagName`**`("tagName")` method returns an array of elements that have the element tag name specified in the parameter. For example:

```
var click2 = document.getElementsByTagName("h1");
```

would return an array of references to all of the <h1> elements in the document. To get access to the first element in the array (with index = 0) we would write:

```
click2[0].onclick = writeNewMessage;
```

Note that as there is only one <h1> in our document, if we were to write **`click2[1]`** we would get an error as the second element does not exist.

This approach means we don't have to modify the HTML element, but rather references the position of the element in the HTML DOM. What is the *disadvantage* of this approach?

[IMPORTANT] Send your tutor the link to your web page running on the Mercury server to be marked off.

Task 2 and Task 3 are optional.

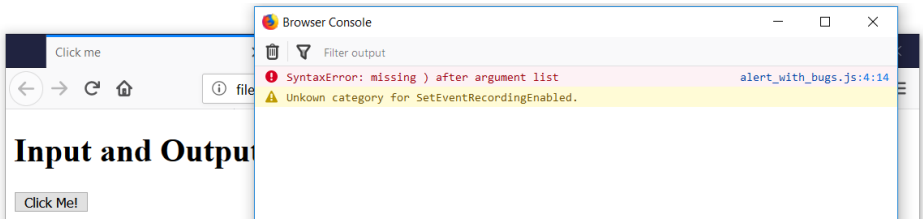
Task 2: Finding *Syntax* Errors in JavaScript (Optional)

Using an HTML or CSS validator will not pick up errors in your JavaScript.
In this task you will use the Firefox *JavaScript Error Console* and *DevTools*.

Bug Hunt

1. Download the files **alert_with_bugs.html** and **alert_with_bugs.js** (in lab05.zip) into your **lab05** folder. Load the file **alert_with_bugs.html** into Firefox. This file references the JavaScript file **alert_with_bugs.js** which has 4 syntax errors in it.
2. Use Firefox to identify error information, such as the line number and the nature of error.
   - Firefox: From the menu (three horizontal lines at the upper right side of the address bar) select "Web Developer" -> "Browser Console

   (Note: the red warning symbol only appears on faulty, parsable code, and persists for the current tab even after the error is solved. Once you fix all errors, you can open your page in a new tab if this bothers you.)



3. Note that not all bugs necessarily will be found at once, as one error might make the rest of the file difficult/impossible to parse.
4. Correct any errors you identify.

Hints:

- Be careful with quotation marks when concatenating string literals and variables
- Single quotation marks must match with single, and double with double.
- JavaScript is CaSe-sEnSiTiVe!
- The error may not always be on the line identified by the error checker but may cascade. This is the case with the last error in the JS file.

Task 3: Finding *Logic* Errors in JavaScript (Optional)

In the previous task we located ***syntax*** errors in JavaScript. Syntax errors result from writing expressions that do not match the grammatical rules of a language – e.g. a variable is not properly declared; a keyword is misspelt; a semi-colon or bracket is missing; etc..

Logic errors, on the other hand, may have perfect grammar but the expressions do not make sense – that is, they do not produce the result we want when the program is executed. Typical examples of logic errors are the incorrect flow of control: statements are in the wrong sequence; selection (e.g. if-then) statements send execution down the wrong path; or loops (e.g. for) iterate the wrong number of times. Other common errors include variables that are referenced when out-of-scope or uninitialized; an incorrect value is assigned to variable; etc..

One way to find logic errors is to trace through the execution of the program as it runs. Debugging tools allow us to:

1. Stop the program execution at any point (set a ***breakpoint***)
2. **Step** though the execution of the program one statement at a time to check if the *flow* of execution is correct.
3. Watch the values of variables change as we step through the program (set a ***watch***).

Step 1: Load a HTML file and view JavaScript in the Debugger

1. Download the files **average.html** and **badaverage.js** (in lab05.zip) into your **lab05** folder. As show below, the Web Page initially displays a button with saying "Enter your first number". The user enters a series of numbers. When the user clicks 'Calculate the Average' the page is supposed to display the sum and average of all the numbers entered.

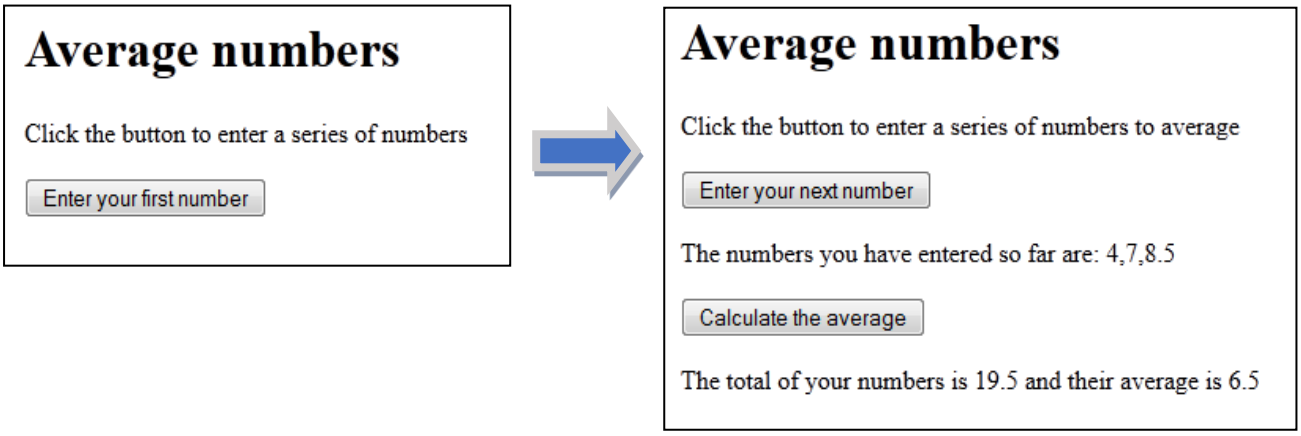
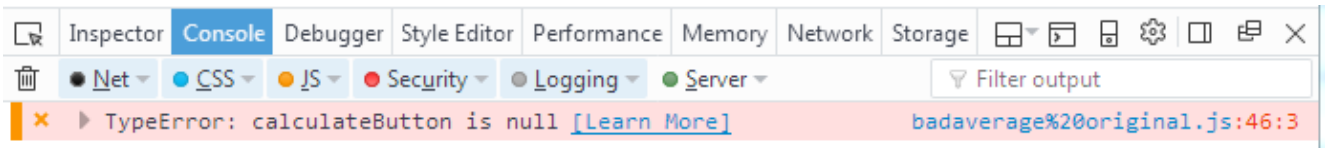


Figure 1 How it *should* work

2. Load the file **average.html** into Firefox.
3. Click on "Enter your first number". ***Notice that nothing happens.***
4. First it is a good idea to check if there are any errors that can be picked up by static analysis of the code. Go to the Firefox menu -> "Web Developer" -> "Web Console" (Cntrl + Shift + K)). This is similar to the console you used in the previous task but displays errors in a pane at the bottom of the Firefox window.
5. Note the error on about line 46. What's wrong? (Hint: remember JavaScript is case-sensitive).

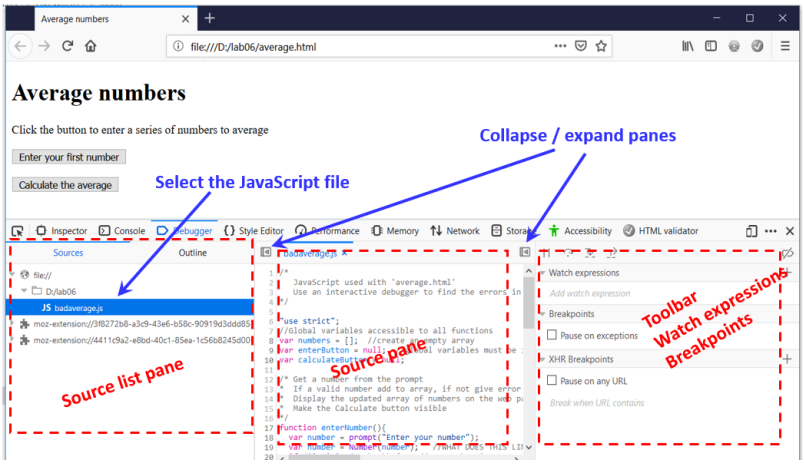


6. Fix the error in Notepad++ (or similar) and save the JS file. Refresh your web page in your browser.
7. In Firefox load the JavaScript debugger:

   Firefox menu (three horizontal lines) -> "Web Developer" -> "Debugger"

   The JavaScript Debugger's user interface is split vertically into three panels: the "Source list pane", "Source pane" and the pane includes toobar, watch expressions, breakpoints and etc.

Use the collapse/expand panes icon to display all the three panes.



8. The JavaScript file referenced in the HTML file should be displayed in the "Source pane" at the bottom of the browser window.

   If the JavaScript file is not displayed in the "Source pane", you can select it from the "Source list pane".

Step 2. Place a Breakpoint on the code and step through it.

In this step you will set a ***breakpoint*** on a line code. When the JavaScript engine in the browser arrives at that line it stops executing and waits for the programmer's input.

We can then step through the code line by line. When we come to a function call in the code we can choose to ***step into*** the code of that function (F11) or ***step over*** it (F10). If we are inside a function and want to return to the code that called the function we can ***step out*** of it (Shift + F11). If we want to ***resume*** automatic execution of the code we can press play (F8).

The controls for the above functions are shown in Figure 2

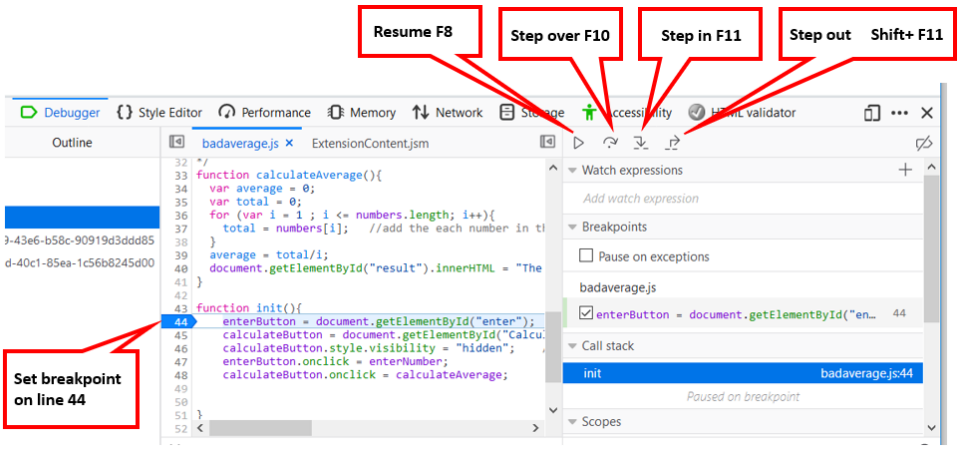


Figure 2 Debugger Controls

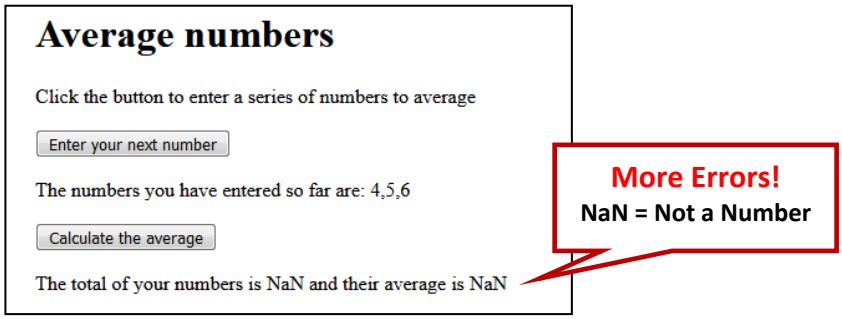
1. Create a *breakpoint* on the first line of the `init()` function by clicking on the line number (44).
2. **Refresh the page.** Step through the function one line at a time by repeatedly pressing F10 or clicking the *Step over* button. (As there are no function calls inside this function you could also press F11). The code is getting references to the buttons on the HTML page, and registering functions to respond to *onclick* events on those elements. It is also hiding the "Calculate" button as we don't have any numbers to calculate yet.

3. When you get to the end of the function statement notice the execution stops. Why? Because it is waiting for input from the user to proceed.

Step 3. Finding errors by watching execution flow

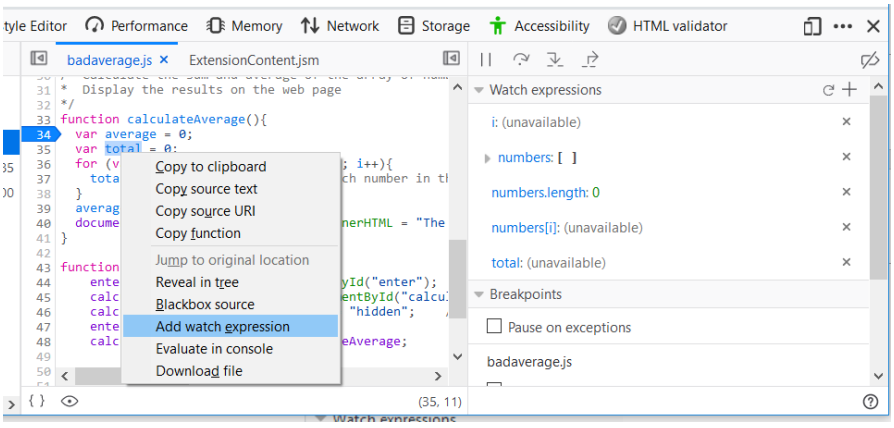
We will now find some runtime logic errors in the code:

1. Click the 'Enter your first number' button, and enter a valid number in the dialog box. This leads to an error message. Click OK.
2. Now enter a non-valid number e.g. "Hi there!". The string you entered or NaN is now added to a message that is supposed to display the numbers entered.
3. In the debugger pane, put a breakpoint on the first executable line of the `enterNumber()` function (line 18). Clear the first breakpoint you created (line 44)
4. Refresh the web page. Notice execution stops at the new breakpoint. Start stepping through the code. Entering a valid number when prompted.
5. Watch as the execution passes through the **`if`** statement. Obviously the condition in the **`if`** statement is wrong.
6. Correct the **`if`** condition in Notepad++ and resave the JS file.
7. Refresh the web page. View the JavaScript in the Debugger window and remove any breakpoints.
8. Enter 3 numbers using the prompt box. Press the 'Calculate the average' button. Although the numbers entered are now correctly displayed, there is something wrong with the calculation.



Step 4. Finding errors using watch expressions

1. Refresh the HTML page, and set a new breakpoint at the start of the `calculateAverage()` function (line 34).
2. We will now set watch expressions on the following variables: `i`, `numbers`, `numbers.length`, `numbers[i]` and `total`. For each of these variables, highlight them, then right-click on the selection to display the popup context menu -> 'Add Watch Expression'. The selected variables to be watched should appear in a pane on the right hand side of the screen.



3. Step through the code watching the values of the variables change inside the **`for`** loop. Your variable watch expressions should look something like the figure below. Notice the array can be expanded to display the values of the individual elements. (**Click the Refresh icon to show the new changes**.)

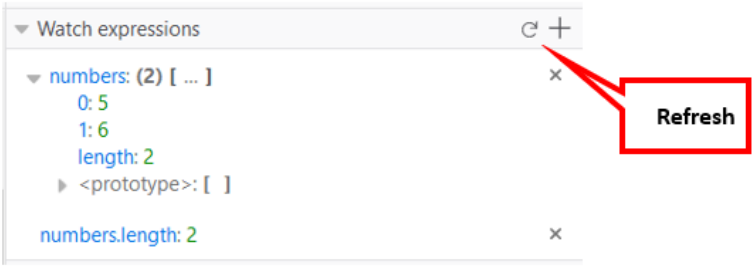
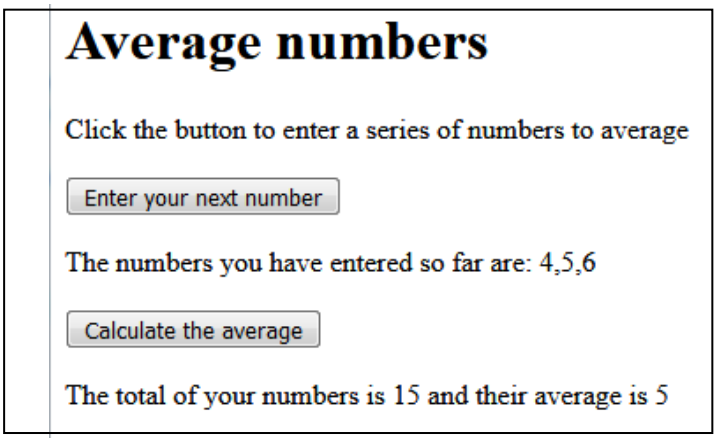


Figure 3 Watch variables

4. Identify at what point `numbers[i]` becomes undefined. What is the value of `i` and `numbers.length` at that point?
5. Using Notepad++, fix the initialise statement and continue condition in the **`for`** loop.
6. Fix any other logic errors you notice in the **`for`** loop. Save the JavaScript.
7. Refresh web page and check that the JavaScript is now working as expected by testing various inputs of numbers.



8. Line 19 looks a bit strange.
```
var number = Number(number);    //WHAT DOES THIS LINE DO?
```
Try removing the line, save and refresh the HTML. Look at the values of the **`numbers`** array in the watch window. What is different ?
9. Reinstate Line 19 and update the comment on Line 19 to explain what it is doing.