HTML5 & Javascript: Lab 1

Objectives of this lab session:

- Become familiar with the WebStorm IDE
- Get to know the debugging features and Javascript console of your chosen browser (the features in Chrome will be described in the lab-sheet – other browsers may differ)
- Write some simple Javascript code to get familiar with the editing features available in WebStorm

Resources required:

- A current web browser (ideally Google Chrome)
- The WebStorm IDE

Part 1: Getting to know WebStorm

WebStorm is an Integrated Development Environment (IDE) that can be used for working in a range of programming languages; Javascript is, however, its specialist subject. You can download and register a copy of WebStorm for use on a home PC or Laptop. See Week 1: Overview on the Moodle site for this module (moodle.uws.ac.uk) for details of how to download, install and register a copy. Please note that the registration will expire at the end of May 2014; a new registration for currently enrolled students will be arranged for beyond that date. Please be aware that under the conditions of the educational licence for WebStorm, you are expressly forbidden from using it for commercial purposes.

This lab-sheet will work on the presumption that you have a properly installed and registered copy of WebStorm. If you prefer to use another IDE, you will need to figure out how (and in many cases, if) it provides the required feature.

Step 1: Running WebStorm and creating a new project

Run WebStorm as follows:

- Double-click on the desktop Icon (Windows, Mac or Linux), or…
- (Windows only) Click the Start button, select All Programs and browse to the WebStorm start-menu entry (usually Jetbrains→WebStorm), or…
- (OS-X only) Press cmd+Space to select spotlight, and type WebStorm, followed by Enter

If WebStorm is properly installed on your system, using any of these actions should result in WebStorm loading – beginning with a splash-screen:

Once the IDE has loaded up, what you will see depends on what WebStorm was doing when it was last closed down. For a new installation, it will open up at the Quick Start page:

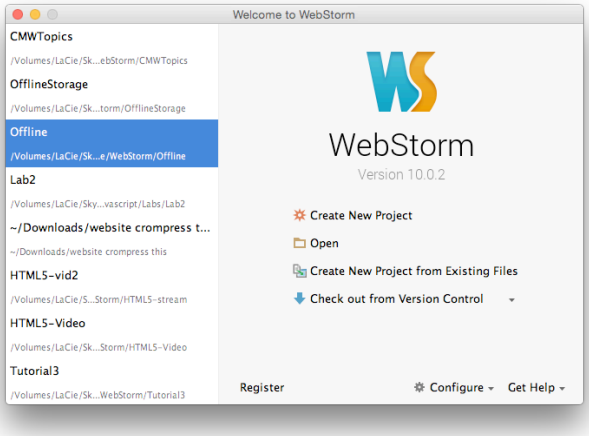


Figure 1: The WebStorm Quick Start page

The top-left option on the Quick Start page is to create a new project. If you are not on the Quick Start page, simply select File→New Project from the main menu. The following dialog opens-up:

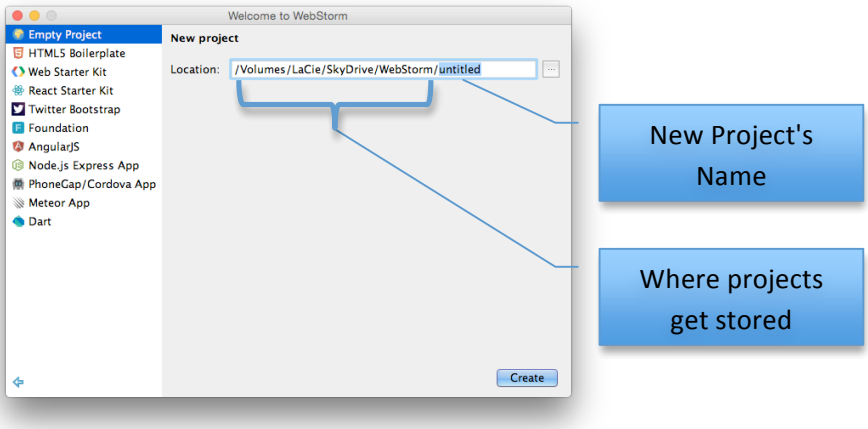


Figure 2: The New Project dialog

To complete this dialog for a new, empty project, simply enter a project name (e.g. Lab1) and specify a location for it. Ideally, this should be on your home drive in a sub-folder with a suitably memorable name – e.g. h:\WebProjects. Note you can press the … button to the right of the Location box to browse to a suitable location. Make sure the type is Empty Project.

Once the New Project command has done its work, it will have created a folder for the project on your home drive (e.g. h:\WebProjects\Lab1) that will contain a hidden folder which will accumulate data about your work. The full WebStorm IDE will open to show your project (currently empty) in the Project window:

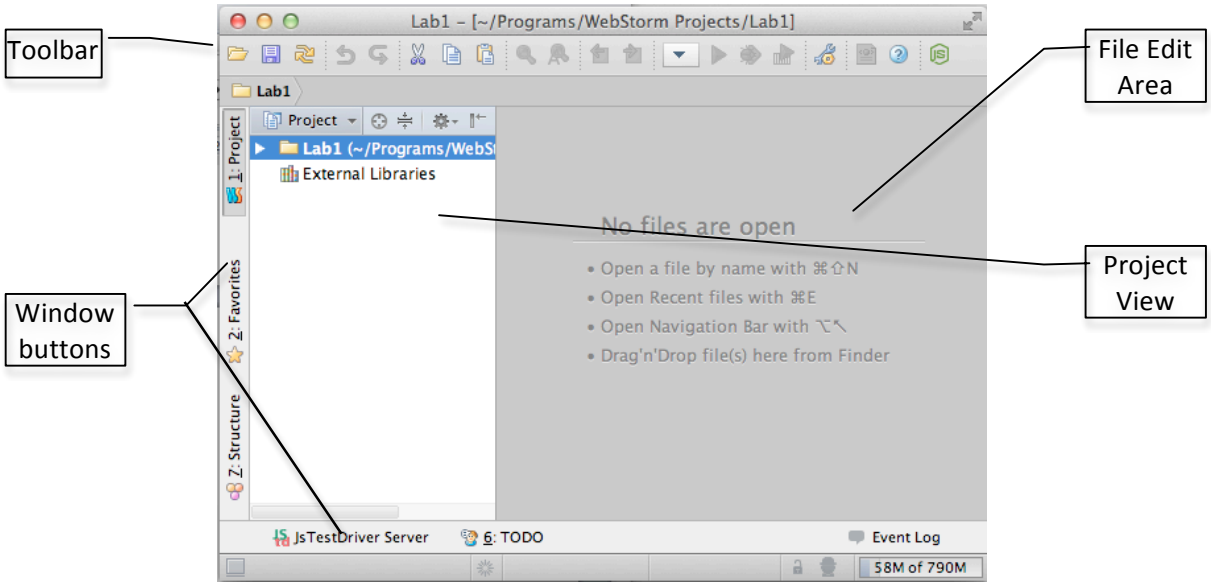


Figure 3: A new project in WebStorm

Project Files

Our new project will contain two files – one HTML file (the host for our Javascript code) and a Javascript file. Add them to the project as follows:

1. Right-click on the Lab1 folder icon in the project view, and select New→HTML File from the pop-up menus. Select HTML file from the Kind: drop-down list, and enter Lab1.html in the name box. A new HTML file will be added to the Lab1 folder, and it will open immediately in the File Edit area. Note that this fill will contain an amount of boiler-plate text – HTML tags for an empty but properly structured document. You can close the file by clicking on the cross at the right of its title tab at the top of the File Edit area, and re-open it by double-clicking on its entry in the Project view
2. Right click again on the Lab1 folder, and this time select New→Directory. It is good practice to separate out Javascript code files into a directory off the main one. Call the new directory JS
3. Right-click on the JS directory, and select New→Javascript file. Name this Lab1.js. The file will open in the edit area, and you should notice that the top of a file has a comment /**Created with Jetbrains WebStorm. …*/. To the left of the comment is a little + sign, and if you click on it the editor will reveal a larger comment block that normally shown in a collapsed state. WebStorm will also allow you to collapse function definitions, object definitions and other comments

Now we have our new project, complete with two files ready to accept HTML and Javascript code.

Editing in WebStorm

The first useful thing we can do is to link the Javascript file to the HTML file. That is done with a simple HTML <script> tag:

```
<script src="js/Lab1.js"></script>
```

To add this script tag, we'll use the WebStorm auto-complete features***, so follow this next bit carefully***:

4. Click at the end of the </title> tag in the HTML file and press Enter to insert a new line before the closing </head> tag. DO NOT type the whole <script> tag yet!

5. Type the opening '<', and notice that as you do, a list of appropriate tag names will pop up in a list
    a. Now type a 's', and it will reduce to a short list of two words beginning with s (script and style). Make sure "script" is highlighted and press the Tab key.
    b. The script keyword will be completed and the cursor will be placed at the end of it. Now type a space, followed by 's'. A list of keywords (the *attributes* that are applicable to <script> tags) should be displayed. Scroll down to the 'src' keyword and press tab – the attribute will be completed apart from the name of the source file (it will look like **<script src="|"** – note the cursor, which will be between the double quotes).

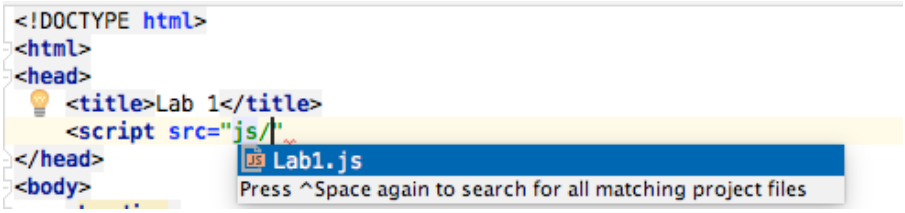


Figure 4: HTML Autocomplete in the WebStorm Editor

    c. Press Ctrl+Space and a list of files and folders will pop-up. Select the JS folder, and JS will be entered between the quote marks. Now enter a / (the folder/file path separator in HTML) and again press Ctrl+Space – a list of the files within the JS folder will pop-up, and since there is only one, you can just press Enter to add it.
    d. Finally, move the cursor to after the closing quote and enter '>' (the closing bracket for the script tag. WebStorm will respond by completing the script tag

- Wherever it can, WebStorm will offer to auto-complete things you are typing. Sometimes, the suggestions made will be not what you want, or you might even find it more bother that it is worth (often for short tags or keywords) – in that case, just keep typing and the pop-up menus will be ignored.

To finish the HTML file, we'll edit some of the existing tags and add some more:

6. Put the cursor in between the opening and closing <title> tags, and enter a title (Lab 1 suits). This title will usually be shown in the browser title-bar, and if you don't enter one, the URL will be shown instead
7. Put the cursor into the <body> section of the page, and enter the opening tag <heading>. You'll be offered several tag types beginning with an 'h' (<h1>, <h2> etc.) although for some reason WebStorm doesn't know yet about the <heading> tag. Regardless, when you type the closing '>' on <heading>, a closing tag will automatically be added
8. Inside the <heading> tag pair, add a <h1> tag (which will trigger an auto-completion), with the text "HTML 5 Test Page" inside the tag. For speed, just type 'h1' followed by Tab

Opening a Page

At this stage, it is worth testing what we have so far. WebStorm gives us a couple of ways of doing this:

- If you move the text cursor up to near the top-right of the html file, a set of icons should appear – each the logo of a browser installed on the machine you're working on. Pressing one will open the page in that browser. Note that this gives you a quick way of testing your web pages on a selection of browsers; note also that it may not pop up if you have a syntax

error on your page – even a very minor one

- You can right-click on the name of the HTML file in the Project View; Open in Browser will be in the context menu (although in this case there will be no choice of browsers)

Using one of these options (there are others that we will examine later), open the page and it should appear as shown in figure 5:

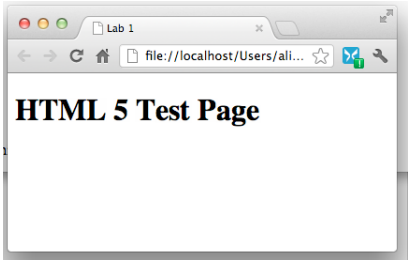


Figure 5: The page in a browser

We'll add a few additional tags to the page to give us some targets for Javascript coding. Add the following tags to the <body> section (new stuff in **bold**), using WebStorm auto-completion where possible, and then test the page by refreshing the browser:

```
<body>
  <heading>
    <h1>HTML 5 Test Page</h1>
    <p id="heading"></p>
  </heading>
  <p id="test"></p>
  <button id="button">Press Me</button>
  <p/>
  Colours: <select id="list">
    <option>Red</option>
    <option>Green</option>
    <option>Blue</option>
  </select>
  <p/>
  Your Birthday: <input type="date" id="dob"/>
  A Number: <input type="range" id="range" min="1" max="10" value="1">
  <span id="value">1</span>
</body>
```

Listing 1: Additional html tags in the page

Using Javascript to Manipulate the Page

Editing Javascript code is much like editing HTML mark-up in WebStorm. Where possible, it will offer to auto-complete what you are typing; sometimes this will be helpful, and at times where it is not, you can ignore the offers.

You should note that several of the mark-up tags you entered into the page html were given "id" attributes. This will allow us to access these tags in our scripting.

We'll start by adding some text to the page as it loads. This is a fairly pointless exercise here, but could be very useful where you wished to add information that was relevant to the status of a page (e.g. the last time the page was visited):

1. Move to the file Lab1.js (you may have to open it in by double-clicking on it in the Project View). We'll add a new function to the page, assigned to the page's onload event:

```
window.onload = function() {
  var para = document.getElementById("heading");
  para.innerText = "A short exercise on creating dynamic web content.";
};
```

   In this code, we are attaching a Javascript function to an event that happens whenever the browser has completed loading the HTML content of a page. Inside the function, a new variable (var) called 'para' is assigned an html element from the document – the one with an id of "heading". This element (a <p>) has its innerText property updated to the text assigned.

   Be careful to add all of the punctuation as shown in the code above, including the semicolons at the end of statements and after the closing curly bracket

2. Move to the browser and re-load the page: the text added to the <p> element should now appear (if not, look for obvious syntax errors – missing closing quote marks etc.)

3. Add the following code inside the window.onload() function. I'll describe the remaining changes to the code in comments within it (*you don't need to type these*):

```
window.onload = function() {
  var para = document.getElementById("heading");
  para.innerText = "A short exercise on creating dynamic web content.";
  var button = document.getElementById("button");      // Get the button element
  button.onclick = function() {                        // Attach a function to it
    alert("I've been clicked");
  };
  var list = document.getElementById("list");
  list.onchange = function() {
    var item = list.options[list.selectedIndex].text; // This code extracts the text
    alert(item + " was selected.");                   // from the selected item in
  };                                                  // list (at the .onchange event)
  var dob = document.getElementById("dob");
  dob.oninput = function() {
    alert("Your birth date is: " + dob.value);        // This picks up the date
  };
  var range = document.getElementById("range");       // This, the slider control
  var value = document.getElementById("value");       // The paragraph after it
  range.onchange = function() {
    value.innerText = range.value;                    // Show the value beside the
  };                                                  // slider.
};
```

4. Having added this code to the JS file, refresh the page and then test each of the controls in turn - press the button, select from the list etc.

You should note several key features of using Javascript coding to access html controls:

- Most html controls (widgets) will provide an ***onclick*** event – a function attached to this will fire when the mouse is clicked on the control
- <select> controls provide an ***onchange*** event, while <input> controls can use ***onchange*** or ***oninput***, depending on what the control type is

- The <input> control for dates can be interacted via the drop-down calendar box (in Chrome only just now), or by typing. If you're entering a date of birth, it is probably easier to select your birthday in the current year and then over-type your year of birth in the text area
- The <input> control for a range does not show the value selected explicitly, so we need to add an extra html element (a is a useful one – it effectively marks a bit of content within a line of mark-up) so that we can change the value as the slider is moved
- Note that all of the event handlers for the controls are created *inside* the ***window.onload*** event handler. Since in Javascript a function is something that can be assigned to a variable, all we need to do is assign an anonymous function definition to the events we wish to handle. We've used a variable to act as the Javascript access point for each html control (button, list, dob, range and value), and these then accept event handlers assigned to them. There are other (more careful) ways of assigning event handlers but in this case we can get away with the simplest.

Creating More General-Purpose Javascript Functions

All of the code we've added so far to the Lab1.js file has been in the form of event-handlers; that is, code that will be called by the browser in response to some occurrence, like the page being loaded or a button being clicked on. It is more often a requirement that we can simply create a Javascript function that can be "called" by other Javascript code as necessary.

1. A function to change the colour of something:

   Since we already have a drop-down selection list for colours, it would be useful to use this to actually affect colours on the page. Here's a function that can do just that:

```
function changeColour(colour){
  var c = 0;
  switch(colour){
    case "Red":
      c = "#f00";  // The code for red
      break;
    case "Green":
      c = "#0f0";  // Green..
      break;
    case "Blue":
      c = "#00f";  // Blue..
      break;
  }
  document.bgColor = c;// Note the U.S. spelling of colour – silly people.
};
```

   If you add the above code to the JS file, it won't change anything. We need to actually call the function to do that. However, we already have an event defined that we can easily turn to that purpose. Simply change the **list.onchange** event handler (inside **window.onload**) to this:

```
list.onchange = function(){
  var item = list.options[list.selectedIndex].text;
  changeColour(item);
};
```

   Now reload the page and test the code again – the page will load to an initial white colour,

but each select you make from that point on should change the background colour of the page. As an exercise, add more colours to the range that can be selected. See http://jdstiles.com/colorchart.html for a complete list (you won't want to do them all).

2. How many days old are you? This isn't an easy question to answer without a calculator, but it's a trivial matter for Javascript:

```
function daysOld(dob){
  var msPerDay = 1000 * 60 * 60 * 24,
    now = new Date(),
    diff = now - dob;
    return diff/msPerDay;
};
```

This function takes a date as a parameter, and works out the difference between that and today's date (new Date()). Unfortunately, Javascript works in milliSeconds, so we need to calculate the factor to divide by (1000mS * 60secs * 60mins * 24hours)

YOUR JOB is to arrange it so that the value worked out in the date-of-birth event handler is passed into this function, and the result of the function is displayed in the alert(). You need to be careful because the value returned by the **dob** date input-box is not a date – its just a string that looks like one (so how do you convert a date string to a date?). As an exercise, amend the code so that it calculates an age in years instead of days (re-name the function accordingly)

3. The slider (an <input> with its type set to "range") can deal with very wide or very narrow ranges of values. For example, you could set min to 0 and max to 255*255*255 (which just happens to be the full range of colours, usually expressed as hexadecimal constants). Alter the coding for the slider value changing so that it can define the background colour for the page.

4. An html document has a .bgColor property. However, all sorts of html elements have various colour settings – see http://www.javascripter.net/faq/changehtmlelementcolor.htm for a good explanation. Experiment with changing the colour of various elements on the page (using Javascript and a combination of controls, like the slider element).

Debugging

It is inevitable that at some stage code you are writing won't work. At this point, you can either stare at the code for hours in the hope of inspiration (this has never worked for me) or try to figure it out by executing the code under the control of a debugger. You have two options to debug a program written in WebStorm:

- Use the WebStorm debugging facilities
- Use the Chrome/Safari built-in debugger or the Firebug plug-in (in Firefox)

I've found the WebStorm debugger to be flaky, although it does had a number of useful features. See the WebStorm HELP page (search for debug) for details.

Using the Chrome Debugger

Using ANY debugger, you need to have some idea where the problem is occurring – for example, you might find that the daysOld() function you wrote earlier is not behaving properly. Finding and fixing the problem is then a matter of stopping the program executing at the entry point of that function, and stepping through the code while watching the values in variables to try to identify the error.

To do this in Chrome (similar for Firebug on Firefox, and the native Safari debugger):

1. Go to the project view in WebStorm, right click on the HTML file and select Open in Browser. Alternatively, you should be able to click on the Chrome icon that appears when you hover the mouse at the top-right of the code editor
2. When Chrome (or your chosen browser) appears, select View→Developer→Developer Tools to open the debugging tools. If you're using a different browser, check on help to find the equivalent menu item
3. Select the Sources button at the top of the window, and the click on the little arrow at the top-left of the Sources window to access the code files (see below)

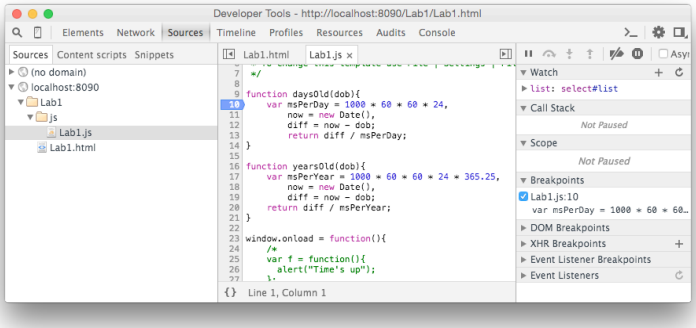


Figure 6: The Chrome debugger showing Sources View, and The Code window with a breakpoint shown (blue arrow).

4. Click on the left hand margin of the code file (in the chrome Sources view) to set a breakpoint next to the first line in the daysOld() function.
5. Now select a date from the Date control on the web page. Execution should stop on the line with the breakpoint, and the step-buttons at the top of the right-hand pane should become enabled.

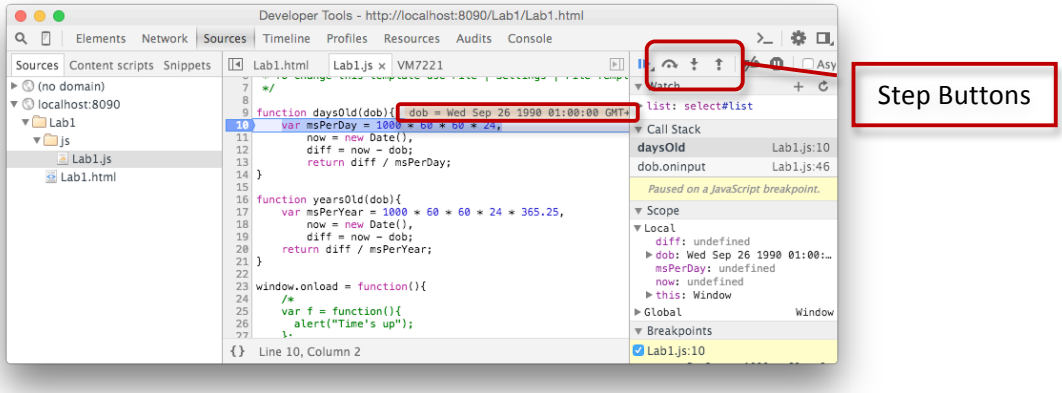
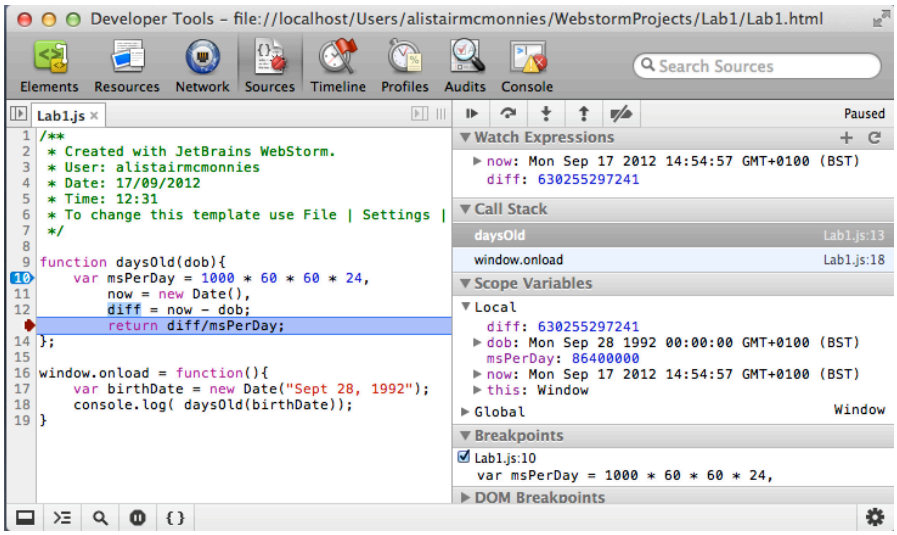


Figure 6: A pop-up tool-tip, and the Step Buttons

6. Note, from above, that if you hover the mouse cursor over a variable name in the code (in this case, the ***dob*** parameter), a window will pop up showing you the variable's contents. If the variable is an object (as here), you can drill into its detailed composition. Note also that if you hover over a variable that has not been assigned a value yet, it will show "undefined"
7. Using the step buttons, step through the code a line at a time. Note the buttons, left to right, are Continue running till the next breakpoint, Step over the current statement, Step into the current statement (which only works if it is a function that is defined in your own

code, Step out of the current function (i.e. execute it till it returns) and Deactivate all breakpoints. At each step, you should hover over variable names to see their current values.

8. Since hovering over a variable name can sometimes be awkward (a single letter variable name is a very small target), you can instead create a Watch entry in the debugger. Right-click on a variable name (e.g. now or diff), and select Add to Watch from the pop-up menu.



Note that as you step through the code, you can watch variable values change. Note that variables that are currently in local scope appear automatically in the Scope Variables pane.

Exercises

1. Go back and look into the execution of the other code you've written for this lab. Place a breakpoint in one of the statements in the ***window.onload*** function and check that you can step into a function that is being called. Note that when you have one function defined inside another (for example, **list.onChange**, which is inside **window.onload**), you need to place a breakpoint accordingly. For example, the line **list.onchange = function(){…** is an assignment of a whole function to the **.onChange** property of the **list** object. Adding a breakpoint to this line will not let you step into the operation of the function, whereas adding a breakpoint to the following line will mean that execution stop only when the list actually changes

2. Have a look at the Developer tools in Safari (Develop Menu→Debug Javascript), and Firefox. In each case they are similar, but you may find you prefer one or other.

End of Lab 1