Building, Testing and Debugging

The shopping list app that was used as an example of a jQuery Mobile program in chapter 5 was unusual in that all of the data in it was on-screen simultaneously. Mainstream program development is normally geared towards building applications where all of the data cannot be on-screen simultaneously; for example, think of the diary app (or calendar app) in your phone. The normal display is a list of entries – the data behind all these entries is only ever accessible one entry at a time. In the vast majority of normal applications, data is displayed as needed. In chapters 6 & 7 we looked at the ways that Javascript could be used to organise data in objects and arrays – structured data. Almost all non-trivial programs work with structured data.

Building an app that involves structured data is one thing; getting it working is altogether more exacting. Current practice in much of the web development industry is to build applications *incrementally*, testing as we go. That way, as we add a new feature and then find it doesn't work as envisaged, we can debug it in relative isolation before moving on to the next new feature. To make this as organised as possible, each feature must be tested as it is added.

It is important to be aware that Testing and Debugging are two different things. Two important definitions are:

- **Testing**: the process of investigating a product to produce evidence of its conformance to its specification
- **Debugging**: methodically exercising and examining a piece of software in order to locate and eradicate defects

These two processes overlap only when a software test indicates the presence of a bug that should then be found and fixed. However, testing is a far more far-reaching activity than debugging, since it is very likely that by properly testing a piece of software the number of defects in it will be reduced. If you like, testing is measuring the quality of a piece of software, debugging is fire fighting.

Debugging with tests

Defects in software take on a lot of different forms. A defect can be anything from the text of a screen being a slightly different colour than you expected to the software being completely inoperative. In some cases defects are even more abstract; for example, if a description of how to operate a particular function is wrong in the manual, this is a software defect even though the actual executable software is error-free.

In a programming module like this one, debugging is usually looked at from a more concrete perspective. We need to learn how to find defects in software whether they were located during formal testing or simply because it didn't do what was expected of it when the user tried to do something. In most cases, we would not differentiate between defects, apart from deciding that some types of defect (e.g. the software doesn't work at all) are more important or severe than others (e.g. the screen background is slightly the wrong shade of blue).

One common way of locating a defect in software is to run it under controlled conditions and examine the values assigned to variables at various points in the execution flow – i.e. apply tests to it. Before the availability of sophisticated debugging software, programmers routinely did this by printing out values in the software as it executed. For example, the function in listing 8.1 has the purpose of returning the surname of a person's full name – e.g. from "Fred Bloggs", the function should extract the name "Fred".

```
function getSurname(fullname){
    return fullname.substring(fullname.indexOf(" "));
}
```

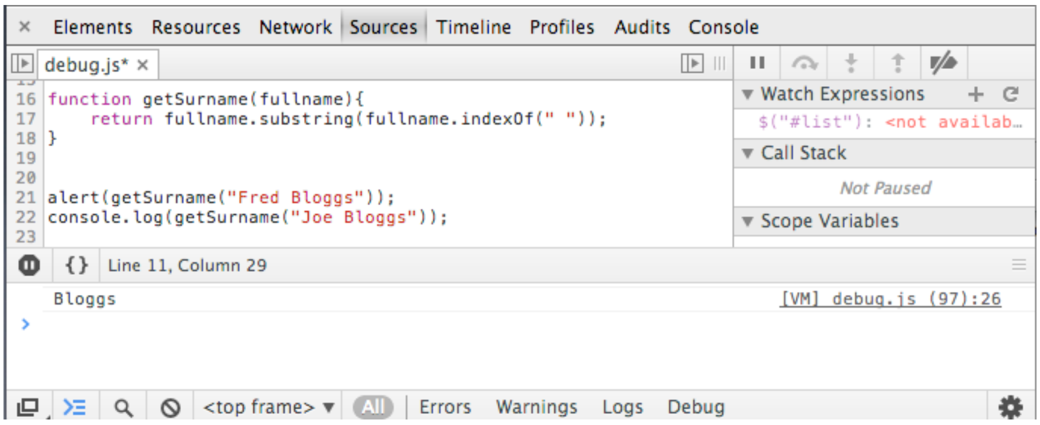
Listing 8.1: A function that needs testing

An easy test of this is to add a function call to the code and print out or display the result in some way. We could use an alert() function call (in a desktop browser), or print the result on the browser console:

```
function getSurname(fullname){
    return fullname.substring(fullname.indexOf(" "));
}

alert(getSurname("Fred Bloggs"));
 // or..
console.log(getSurname("Fred Bloggs"));
```

Listing 8.2: A simple test (in two forms) of the function

In either case, our initial impression might be that the function works. However, a *very* close examination of the output on the console indicates a problem:



```
function getSurname(fullname){
    return fullname.substring(fullname.indexOf(" "));
}


alert(getSurname("Fred Bloggs"));
console.log(getSurname("Joe Bloggs"));
```

```
 Bloggs                                        [VM] debug.js (97):26
```

Figure 8.1: The Console output of our test

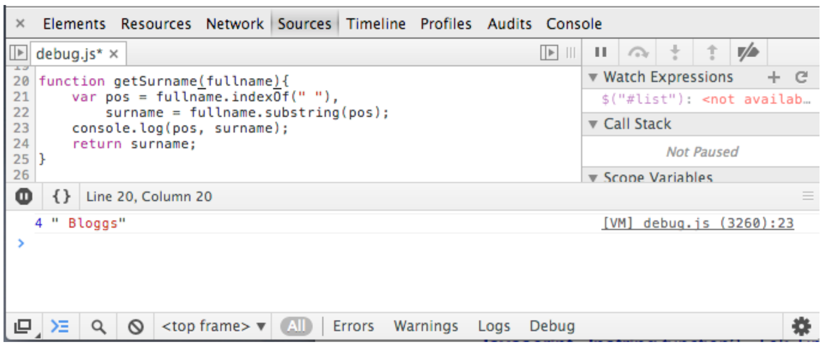
Examining the output in figure 8.1, you can just about make out that the name "Bloggs" has been indented by one character – i.e. our function returns the surname with a leading space (" Bloggs"). How does this come about? We can check by making the working of the function a bit more explicit; the function relies on the expression `fullname.indexOf(" ")` giving us the position of the first character in the surname (I'm hoping that by now you already know what the problem is, but bear with it for a bit – the answer is much less useful than a description of the process we can always use to find it in these notes). Part of the "problem" for debugging this code is that the function is defined in such a way that an important bit of information is hidden away in a function call – the statement:

```
return fullname.substring(fullname.indexOf(" "));
```

This effectively works out two key pieces of data (the index of the start of the surname and the actual surname) in a single shot. It would be useful for intermediate values in calculations to be available during the debugging process. We can revise the function definition to make this possible:

```
function getSurname(fullname){
    var pos = fullname.indexOf(" "),
        surname = fullname.substring(pos);
    console.log(pos, surname);  // We now get to see the values
    return surname;
}
```

Listing 8.3: 'Unpacking' the workings in a function



```
function getSurname(fullname){
    var pos = fullname.indexOf(" "),
        surname = fullname.substring(pos);
    console.log(pos, surname);
    return surname;
}
```

```
$("#list"): <not availab…
```

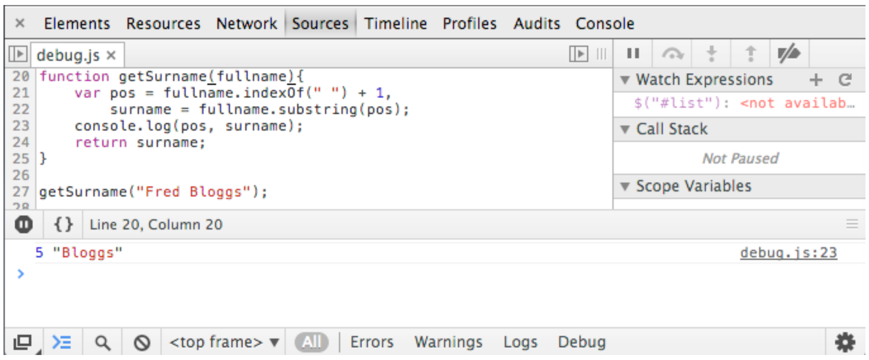
```
4 " Bloggs"                                        [VM] debug.js (3260):23
>
```

Figure 8.2: Testing the revised function

The version of the function in listing 8.3 does two new things – first, it makes the value of the expression `fullname.indexOf(" ");` available in a variable. Secondly, it prints out the value on the console, along with the value that the function returns. We now ought to be able to work out exactly what's happening – instead of the position of first character in the surname, the `indexOf()` function is returning the position of the space before the surname (F-r-e-d-<space>, counting from 0). In fact the console display makes this much clearer now as a side effect of us displaying TWO variable values instead of one (the space at the start of " Bloggs" is much more obvious because of the quote marks), although this is just a happy accident and not something we'd rely on. We can now simply add 1 to the index to get the true start of the surname:

```
function getSurname(fullname){
    var pos = fullname.indexOf(" ") + 1,
        surname = fullname.substring(pos);
    console.log(pos, surname);
    return surname;
}
```

Listing 8.4: A better version



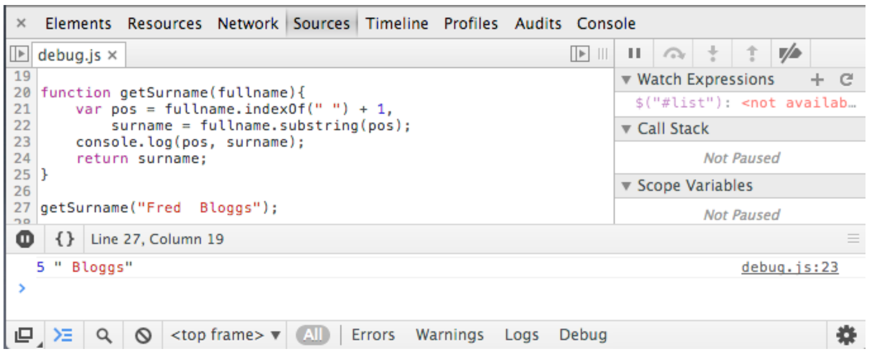
```
function getSurname(fullname){
    var pos = fullname.indexOf(" ") + 1,
        surname = fullname.substring(pos);
    console.log(pos, surname);
    return surname;
}

getSurname("Fred Bloggs");
```

```
$("#list"): <not availab…
```

```
5 "Bloggs"                                         debug.js:23
>
```

Figure 8.3: The result in the console.

So, can we now add that function to our system and assume that it will always do the right job? Look at this test and decide for yourself:



```
function getSurname(fullname){
    var pos = fullname.indexOf(" ") + 1,
        surname = fullname.substring(pos);
    console.log(pos, surname);
    return surname;
}

getSurname("Fred  Bloggs");
```

```
$("#list"): <not availab…
```

```
5 " Bloggs"                                   debug.js:23
```

Figure 8.4: What's gone wrong here? (look at the spacing in the function call)

What this tells us is that basic testing is not enough. All we need is a user who adds more than one space between their first and second names by accident or design and the function misbehaves. Incidentally, to see why this might be important, just consider that the full name could be what a user enters (e.g. into their address book) and that we would want to display entries in the address book in alphabetical order of surname. A name that begins with a space would not get sorted into the correct position in the alphabetical listing, and this is a bug.

The final fix for this would be to change the coding of the function so that it used the expression:

```
pos = fullname.lastIndexOf(" ");
```

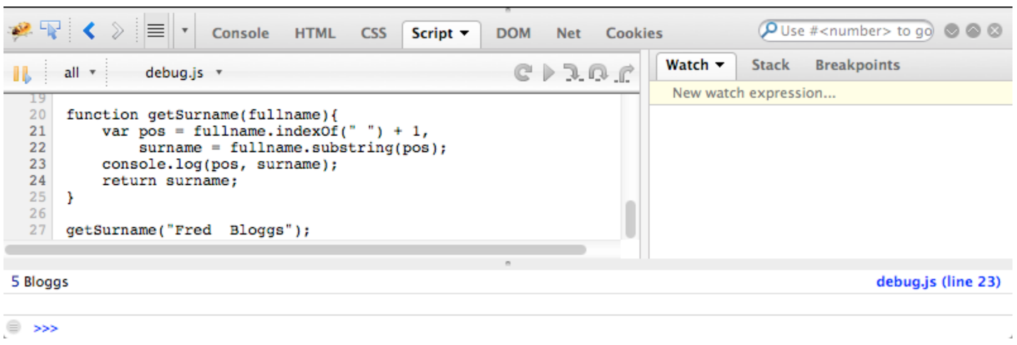
The `lastIndexOf()` string function in Javascript is perfect for this situation. We would now get proper surnames returned for ***most*** of the variants that a user might enter. However, if anyone entered "Bloggs, Fred" or "Fred Bloggs Jnr." or "Fred Bloggs esq." or anything similar, all bets are off. There is virtually no level of programmer diligence that a perverse end-user could not subvert in some way.

Debugging with Tools

It is all very well adding extra statements to a project to enable debugging, but extra statements do affect the performance of your program. While a single `console.log()` statement would not add a noticeable penalty, imagine the impact of placing one or two of them inside code that did a complex animation task inside a loop that executes 50-100 times per second for each game character on the screen; suddenly these statements start to have a major impact on the code.

In some very switched on development regimes, `console.log()` statements are automatically stripped from a program before it is distributed, and that is a very useful thing to do. However, instead of inserting `console.log()` statements into our code, we can use available tools to examine the code as it executes; generally this is a more flexible way to debug code. Best of all, the tools are free.

The Firefox browser can be retro-fitted with a plug-in called Firebug, which provides a very powerful set of developer tools including a Javascript source code debugger, HTML element inspection, CSS rule inspection and editing, DOM inspection and logging and timing of network requests. Figure 8.5 shows Firebug's combined Source code and Console view:



```
function getSurname(fullname){
    var pos = fullname.indexOf(" ") + 1,
        surname = fullname.substring(pos);
    console.log(pos, surname);
    return surname;
}

getSurname("Fred  Bloggs");
```

```
5 Bloggs                                                debug.js (line 23)
>>>
```

Figure 8.5: The Firebug developer tools

You will notice that the Firebug developer tools are not very different from the tools built in to Google Chrome (shown in figures 8.1 to 8.4). The only real difference is that the Google Chrome tools are built in to the browser and so no plug-in is required. You will find similar tools built into Internet Explorer (from Version 9 on) and Safari. My personal preference is for the Chrome tools, but that's mostly just because I'm just more used to using them than for any real technical reasons.

Where debugging tools like these come to the fore is when you are trying to find a defect in a much bigger application; you may only be aware that a page has not displayed properly, but have no idea where, among thousands of lines of code, the problem comes from. In that case, browsers provide the "big guns" for debugging.

Lets look at an example. The app will be a simple phone book app (which will be useless since any phone that can run a web app will already have a phone book – no matter). In this well see some more of the power of jQuery Mobile as we go along. This is incidental, since the real purpose will be to show in a more realistic setting how developer tools can be used to de-bug an application. Every error that gets introduced into this app is one that I've been plagued with at some point in the past, so apart from starting a new project from scratch on your own, this is about as detailed a debugging exercise as it gets.

At each stage of this app, we'll introduce some functionality, test it using a few test cases and then try to find and remove bugs – I'll have deliberately incorporated these bugs, but check out the previous paragraph for justification. We'll start with the basic user-interface – this is entirely in jQM HTML and will (I hope) be bug-free. It is not that bugs in HTML as less important, but they are usually more obvious when you view the page. Finding bugs in Javascript code is where there is a real potential for headaches.

The Phone Book User-Interface

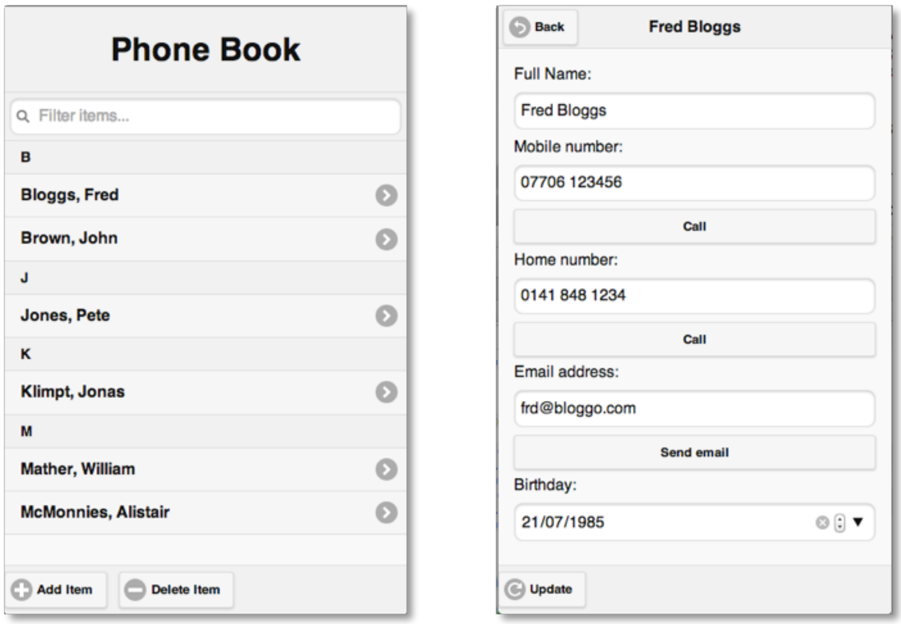


Figure 8.6: The basic HTML user-interface for our Phone Book App

Figure 8.6 shows a fairly complex looking user-interface for a mobile web app. However, apart from one very short piece of Javascript (which generates the 'letter' dividers between the names in the list), this is all done using standard jQM mark-up. We will of course need to provide a fair amount of Javascript functionality (for adding and removing entries, to define a data structure to store entry data, and to manage the stored data while the app is not running). However, jQM mark-up takes care of much of the core functionality, including the Search/Filter box at the top of the Phone Book page, the operation of the Call buttons on the entry page, transitions between pages etc.

The full mark-up for what you see in figure 8.6 is in listing 8.5. From here, we'll need to start writing Javascript code and de-bugging it as we go.

```
<!DOCTYPE html>
<html>
<head>
    <title>Phone Book App</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet" href="http://code.jquery.com/mobile/1.2.1/jquery.mobile.min.css" />
    <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
    <script src="http://code.jquery.com/mobile/jquery.mobile-1.2.1.min.js"></script>
    <style type="text/css">
        .img {padding-left:70px; }
        h1 {text-align: center;}
    </style>
</head>
<body>
    <div data-role="page" id="main">

        <div data-role="header">
            <heading>
                <h1>Phone Book</h1>
            </heading>
        </div>
```

Scripts & simple CSS Rules

Main Page

```
        <div data-role="content">
            <ul id="list" data-role="listview" data-filter="true" data-autodividers="true">
                <li><a href="#">Bloggs, Fred</a></li>
                <li><a href="#">Brown, John</a></li>
                <li><a href="#">Jones, Pete</a></li>
                <li><a href="#">Klimpt, Jonas</a></li>
                <li><a href="#">Mather, William</a></li>
                <li><a href="#">McMonnies, Alistair</a></li>
            </ul>
        </div>

        <div data-role="footer" data-position="fixed">
            <nav>
                <a href="#" data-role="button"
                   data-inline="true" data-mini="true"
                   data-icon="plus" id="add">Add Item</a>
                <a href="#" data-role="button"
                   data-inline="true" data-mini="true"
                   data-icon="minus" id="del">Delete Item</a>
            </nav>
        </div>
    </div>
```

Dummy Data

This will be removed from the final app

```
    <div data-role="page" id="entry">

        <div data-role="header">
            <h1 id="name">Fred Bloggs</h1>
            <a href="#" data-rel="back" data-icon="back">Back</a>
        </div>

        <div data-role="content">
            <form>
                <label for="fullname">Full Name:</label>
                <input type="text" id="fullname"/>
                <label for="mobile">Mobile number:</label>
                <input type="tel" id="mobile"/><a data-role="button" data-mini="true"
                       href="tel:07706123456">Call</a> <!—Note – dummy data for now-->
                <label for="tel">Home number:</label>
                <input type="tel" id="tel"/><a data-role="button" data-mini="true"
                       href="tel:07706123456">Call</a> <!—Note – dummy data for now-->
                <label for="bday">Birthday:</label>
                <input type="date" id="bday"/>
            </form>
        </div>

        <div data-role="footer" data-position="fixed">
            <a href="#" id="update" data-role="button" data-icon="refresh">Update</a>
        </div>
    </div>
</body>
```

Entry Page

```
<script>
    // Note – this script inserts letter dividers into the "#list" element on the
    // main page.
    $(document).on("pageinit", "#main", function(){
        $("#list").listview({
            autodividers: true,
```

Script to insert 'letter' dividers

```
            autodividersSelector: function (li) {    // Do for each entry
                var out = li.text().substring(0, 1); // Use initial letter of surname
                return out;
            }
        }).listview('refresh');
    });
</script>
</html>
```

Listing 8.5: The Complete mark-up for the Phone Book User Interface (includes some dummy data)

Note that while the HTML in listing 8.5 incorporates a <script> element, this would be moved into a separate JS file in the final version. Note also that the dummy data that is in place just now so that you can see the effects of the mark-up will be removed in the final app – the only entries the app will display and work with will be the ones the user enters.

Scripts – data and functions

We're now ready to start adding Javascript code to this app. We have two different areas to deal with:

1. How to represent the information in the Phone Book Entries
2. How to manipulate the list of phone entries

The first of these is simple enough – we'll use an object to store the bits of data for each entry, and manage an array of entries in the application to keep our phone-book data neatly encapsulated. The second will give us a bit more work:

- When we add/delete/update entries, we will manipulate the array of objects directly. That means we'll need to incorporate a way to synchronize what is seen on the screen with what data there is in the array
- When the user selects an entry to interact with, we'll need to retrieve the relevant array member and display its details on the Entry page. If the "update" button is pressed, we'll need to update that member of the array – we'll also need to re-synchronise the array data with the main page in case a name is changed
- Adding a new entry will involve creating an empty object, adding it to the array and then treating it as an update
- Deleting an entry will be done by simply removing the entry from the array and then re-synchronising the data with the display

So here is a first cut at the definition of an Address Book entry:

```
var Entry = function(name, mobile, tel, email, dob) {
 this.name = name;
 this.mobile = mobile;
 this.tel = tel;
 this.email = email;
 this.dob = dob;
}
Entry.prototype.displayName = function() {
 var firstnames, surname;
 firstnames = this.name.substring(0, this.name.indexOf(" "));
 surname = this.name.substring(this.name.lastIndexOf(" ") + 1);
 return surname + ", " + firstnames;
}
```

```
Entry.prototype.isBirthday = function() {
  var bday = this.dob;
  bday.fullYear = new Date().fullYear;
  if(bday.getDate() === new Date().getDate()) {
    return true;
  } else {
    return false;
  }
}
Entry.prototype.changeName = function(firstnames, surname){
  this.name = firstnames.trim() + " " + surname.trim();
}
```

Listing 8.6: The Entry object type

Note that we've added some prototype functions to the Entry type; to get the name arrayed as it will be displayed in the list (so we can put them in alphabetical order), to return whether today is the Entry's birthday and to change the Entry's name (since we need to maintain a very strict organisation to make sure spaces don't upset the ordering). We won't need similar functions to change telephone numbers or email addresses because we don't have the need to organise these in order.

Given the definition of an individual Entry object, we can now consider how to manage an array of them:

```
var entries = [];     // This will accommodate all of the phonebook entry data

function addEntry(name, mobile, tel, email, dob) {
  var e = new Entry(name, mobile, tel, email, dob); // Create the entry
  entries.push(e);                                  // Add to the array
  sortEntries();                                    // Re-order the array
}

function removeEntry(name){
  var pos = -1, index;
  for(index = 0; index < entries.length; index += 1){
    if(name === entries[index].displayName()) {
      pos = index;
      break;
    }
  }
  if(pos > -1) {
    entries.splice(pos, 1);
  }
}

function sortEntries() {
  entries.sort(function(a, b) {
    if(a.displayName() < b.displayName()){
      return -1;     // indicates that
    }
    if(a.displayName() > b.displayName()) {
      return 1;
    }
    return 0;
  }
}
```

```
function entryList(){
  var index, list = "";
  for(index = 0; index < entries.length; index += 1){
     list += "<li><a href='#entry'>" + entries[index].displayName() + "</a></li>";
  }
  return list;
}
```

Listing 8.7: The array of entries, and key functions for it

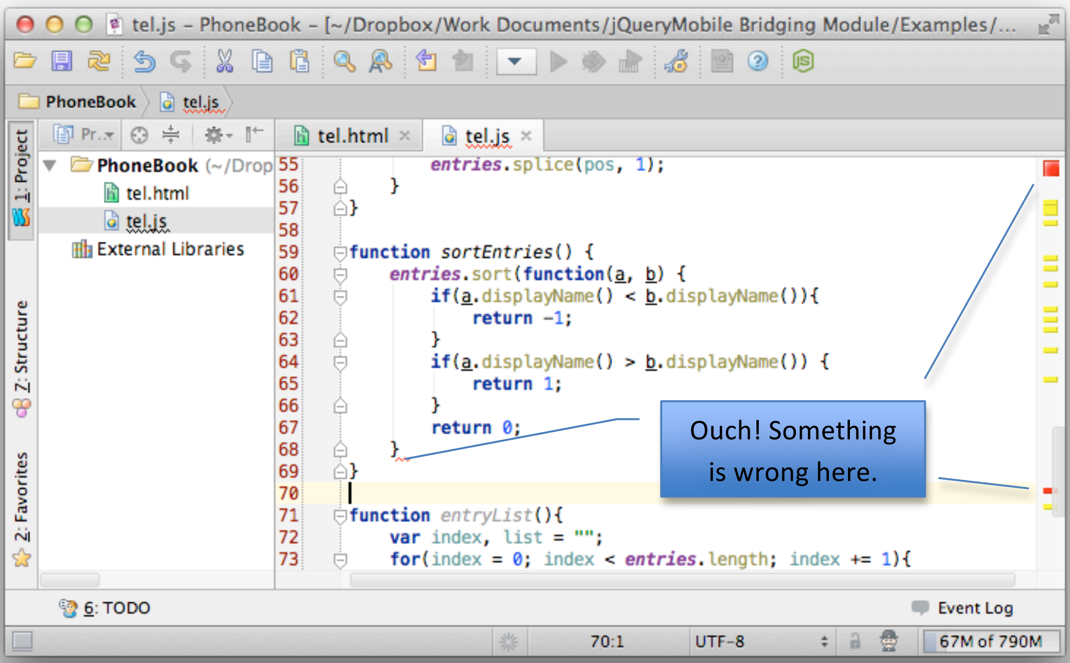
Note that some important functions have been defined for the array of entries, these being:

• addEntry()	this has an obvious purpose, although you should note that after each addition the whole array is sorted into order. This is pretty inefficient, but do-able for the number of entries in a normal phone book
• removeEntry()	this also has an obvious purpose. Note the process – find where the entry is in list and if it exists, use the `splice()` method to remove it. We test for matches in `displayName()`, since an entry would be deleted from the main screen, where the list contains only displayed names
• sortEntries()	In Javascript, arrays have a `sort()` method that can normally be used to sort arrays into obvious order (alphabetical, numerical). However, the objects in our array are arranged in order of `displayName()`, so we need to provide a function that tells the sort() method this. Basically, returning -1 (or a number less than 0) says that the order of the parameters is first then second; returning a positive number says the order is second then first. Returning zero means that there is no difference in sort order between the first and second (e.g. two "John Smith"s)
• entryList()	This function will make the user-interface processing a lot easier. We need to display our entries as a sequence of `<li>` elements (the total HTML of which can be put into the `<ul>` element. This function simply joins the `displayName()` properties of all of the entries up as a sequence of `<li>` elements. Note that the entry includes an anchor tag to take the app to the entry page – we'll be able to use that to display the selected entry – see later.

At last – something to test

We're now in a position to incorporate this code into our app and test it. In the early stages, tests should be done with as little reference to the user-interface as possible, since this could affect the tests in ways we don't yet understand. Remember, our purpose here is to find defects if there are any (there are!), and so simply adding a bunch of entries without being deliberately awkward will not be enough.

First – save the code in our Javascript file (tel.js) and add a reference to this script file to the HTML file. With the code in WebStorm, without even trying to execute it, we get the first hint that there is a problem:

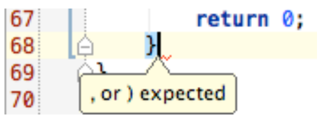


```
            entries.splice(pos, 1);
        }
}

function sortEntries() {
    entries.sort(function(a, b) {
        if(a.displayName() < b.displayName()){
            return -1;
        }
        if(a.displayName() > b.displayName()) {
            return 1;
        }
        return 0;
    }
}

function entryList(){
    var index, list = "";
    for(index = 0; index < entries.length; index += 1){
```

Figure 8.7: WebStorm has found a problem

Note that WebStorm makes it very clear there is an issue in code (this is due to the excellent integration with jsLint, which is the standard Javascript code quality tool). The big square red block at the top of the right-hand margin says, "there is at least one error in this code". The small red rectangle lower down shows us roughly where the error is in the file. Click on this any the text cursor will move directly to the offending line.

Finally, the red squiggly underline shows us exactly where the error is. Hover the mouse cursor over it and a tool-tip will pop-up:



```
        return 0;
    }
}
```

, or) expected

Figure 8.8: A tool-tip, explaining the error

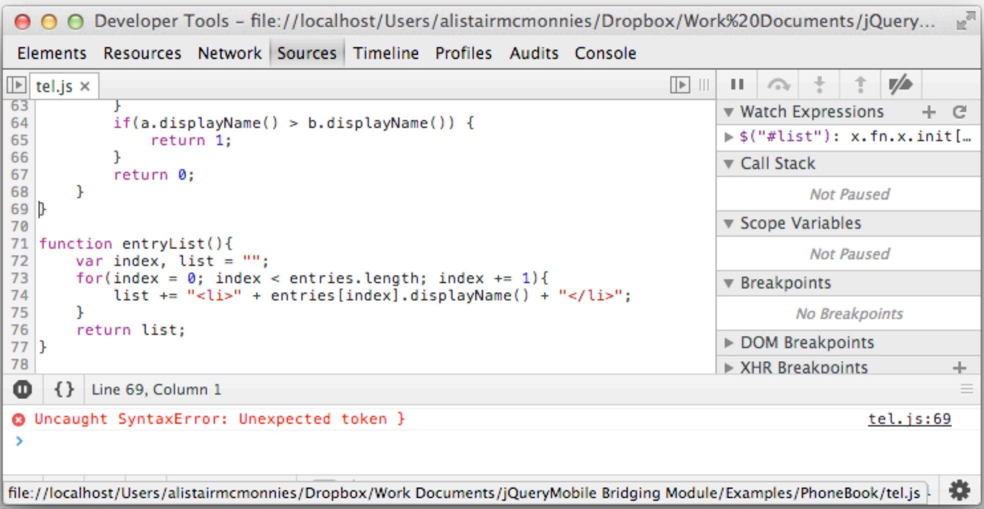
To explain this, consider that the `sortEntries()` function calls the array function `array.sort()`, which *may* include a function definition to indicate how the sort order should be defined (see `sortEntries()` in the above bullet list, or check the page at http://www.w3schools.com/jsref/jsref_sort.asp for more details). We've passed the definition of an anonymous function into `array.sort()`, and it is easy to get mixed up with which brackets to close in this situation. The sortEntries() definition should actually be:

```
function sortEntries() {
 entries.sort(function(a, b) {
   if(a.displayName() < b.displayName()){
     return -1;
   }
   if(a.displayName() > b.displayName()) {
     return 1;
   }
   return 0;
 });
}
```

These brackets should enclose a whole function definition

Listing 8.8: Yay! For WebStorm – it pointed out the missing closing bracket

Note that if WebStorm had not identified this error, the Chrome Developer tools (or Firebug, or any other browser's tools for that matter) would indicate the problem in a console window as soon as the page was loaded:


Developer Tools – file://localhost/Users/alistairmcmonnies/Dropbox/Work%20Documents/jQuery…
Elements Resources Network Sources Timeline Profiles Audits Console
tel.js ×
Watch Expressions
Call Stack
Not Paused
Scope Variables
Not Paused
Breakpoints
No Breakpoints
DOM Breakpoints
XHR Breakpoints

```
63         }
64         if(a.displayName() > b.displayName()) {
65             return 1;
66         }
67         return 0;
68     }
69 }
70
71 function entryList(){
72     var index, list = "";
73     for(index = 0; index < entries.length; index += 1){
74         list += "<li>" + entries[index].displayName() + "</li>";
75     }
76     return list;
77 }
78
```

```
$("#list"): x.fn.x.init[…
```

```
Line 69, Column 1
Uncaught SyntaxError: Unexpected token }                    tel.js:69
>
file://localhost/Users/alistairmcmonnies/Dropbox/Work Documents/jQueryMobile Bridging Module/Examples/PhoneBook/tel.js
```

Figure 8.9: Chrome's console window pointing out the error

In the console window, we can see "Uncaught SyntaxError: Unexpected token }" – at the end of the line we're told the file the error was in and the line number (tel.js:69). Simply click on that link to highlight the error.

So, without doing any serious de-bugging, we've identified a problem in the code already. As a general principle:

- Use the WebStorm editor to identify errors and remove them as soon as possible. That will mean the code that you load into the browser will be as error free as you can get it before you start testing
- Sometimes WebStorm will miss errors, and so the next step is to load the app into a browser and check the console window for error messages
- Some error types will not show up. Specifically, errors that can't be identified until the browser tries to run the code. That's when the real de-bugging starts.

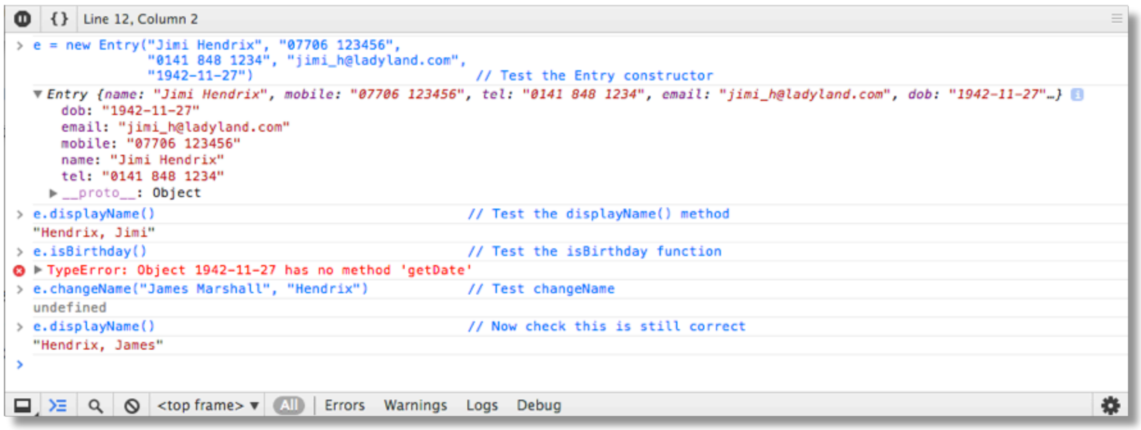
When debugging Javascript code, it is usually a good idea to start with a simple checklist of things we need to test:

1. Does the Entry constructor function correctly create an entry?
2. Does an Entry object properly return its displayName()?
3. Does an Entry object correctly report on whether today is the entry's birthday?
4. Can I change an Entry's name (does this reflect properly in the displayName()?
5. Once the JS code is loaded, does the entries[] array exist?
6. Does addEntry() correctly add a new entry to the array?
7. Does deleteEntry() correctly remove an entry?
8. What happens if we try to remove a non-existent entry?
9. Is the list of entries correctly sorted into order?
10. Does the entryList() function generate correct and accurate HTML?

We can use the Chrome console to answer these questions. To keep things as clear as possible, questions 1-4 specifically test the new Entry type and questions 5-10 test the entries array.

Testing the Entry type

Figure 8.10 shows the tests on the Entry type:



```
> e = new Entry("Jimi Hendrix", "07706 123456",
                "0141 848 1234", "jimi_h@ladyland.com",
                "1942-11-27")                        // Test the Entry constructor
  ▼ Entry {name: "Jimi Hendrix", mobile: "07706 123456", tel: "0141 848 1234", email: "jimi_h@ladyland.com", dob: "1942-11-27"…}
      dob: "1942-11-27"
      email: "jimi_h@ladyland.com"
      mobile: "07706 123456"
      name: "Jimi Hendrix"
      tel: "0141 848 1234"
    ▶ __proto__: Object
> e.displayName()                                    // Test the displayName() method
  "Hendrix, Jimi"
> e.isBirthday()                                     // Test the isBirthday function
  ▶ TypeError: Object 1942-11-27 has no method 'getDate'
> e.changeName("James Marshall", "Hendrix")          // Test changeName
  undefined
> e.displayName()                                    // Now check this is still correct
  "Hendrix, James"
>
```

Figure 8.10: Using the console to test the Entry object type

As you can see, the simple tests run on the Entry type have shown up a couple of problems. A new Entry does seem to be properly created, as shown by the first command (`e = new Entry(…)` and its result, shown below it). The displayName() function properly re-formats the name as "Hendrix, Jimi".

However, the function to indicate whether today is the entry's birthday is not behaving, and produces the console error report "**TypeError: Object 1942-11-27 has no method getDate()**". This tells us that "1942-11-27" is not recognised by Javascript as a date. That is simply common sense – anything between quotes in Javascript is interpreted as a String. There are two possible solutions – the first is that we pass a proper date into the `Entry()` constructor function, for example:

```
E = new Entry("…", "…", "…", "…", new Date("1942-11-27"));
```

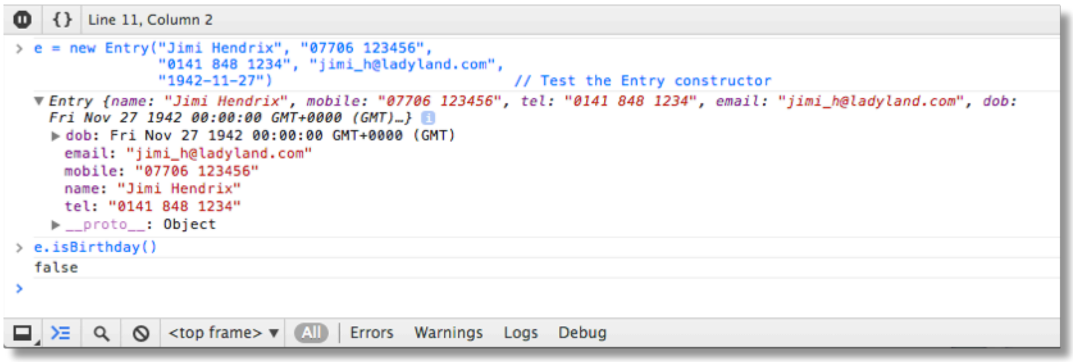
By calling the `Date()` constructor, the input data will either be converted to a date object or will generate an error (e.g. if we passed `new Date("1942-110-35")`). While generating an error might seem inconvenient, it is a better option than allowing invalid data into a new object. However, when we build a new type, we should see it as an obligation to make it as easy to use as possible. I'd advise an alternative approach, which would be to accept the input date as a string, and use the `Date()`

constructor *inside* the constructor function:

```
var Entry = function(name, mobile, tel, email, dob) {
    this.name = name;
    this.mobile = mobile;
    this.tel = tel;
    this.email = email;
    this.dob = new Date(dob);    // Convert a text date to a Date object
};
```

Listing 8.9: A revised Entry() constructor

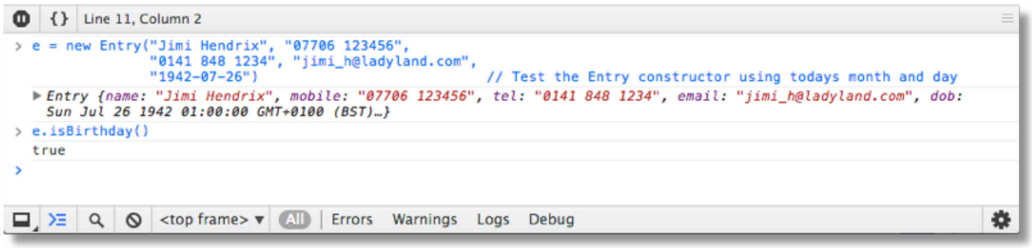
We can repeat the tests now to check whether this fixes the problem:



```
> e = new Entry("Jimi Hendrix", "07706 123456",
                "0141 848 1234", "jimi_h@ladyland.com",
                "1942-11-27")                            // Test the Entry constructor
  ▼ Entry {name: "Jimi Hendrix", mobile: "07706 123456", tel: "0141 848 1234", email: "jimi_h@ladyland.com", dob:
    Fri Nov 27 1942 00:00:00 GMT+0000 (GMT)…}
    ▶ dob: Fri Nov 27 1942 00:00:00 GMT+0000 (GMT)
      email: "jimi_h@ladyland.com"
      mobile: "07706 123456"
      name: "Jimi Hendrix"
      tel: "0141 848 1234"
    ▶ __proto__: Object
> e.isBirthday()
  false
>
```

Figure 8.11: Testing the revised constructor

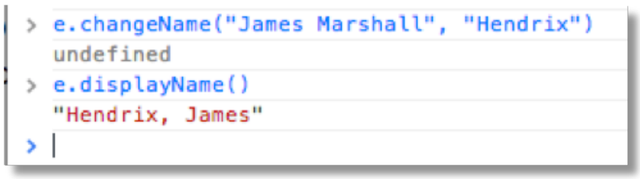
Note that we can see even before the test of the `isBirthday()` function that the object created contains a proper date, since the console displays it as a complete date and time (*`Fri Nov 27 1942 00:00:00 GMT+0000 (GMT)`*). We can do a further test on the `isBirthday()` method by creating a new Entry() object who's birthday is today (26th July at time of writing):



```
> e = new Entry("Jimi Hendrix", "07706 123456",
                "0141 848 1234", "jimi_h@ladyland.com",
                "1942-07-26")                          // Test the Entry constructor using todays month and day
  ▶ Entry {name: "Jimi Hendrix", mobile: "07706 123456", tel: "0141 848 1234", email: "jimi_h@ladyland.com", dob:
    Sun Jul 26 1942 01:00:00 GMT+0100 (BST)…}
> e.isBirthday()
  true
>
```

Figure 8.12: A further test that confirms the isBirthday() function

The final test of the Entry type is whether we can change the `Entry's` name and whether the new name is reflected in the `displayName()`:



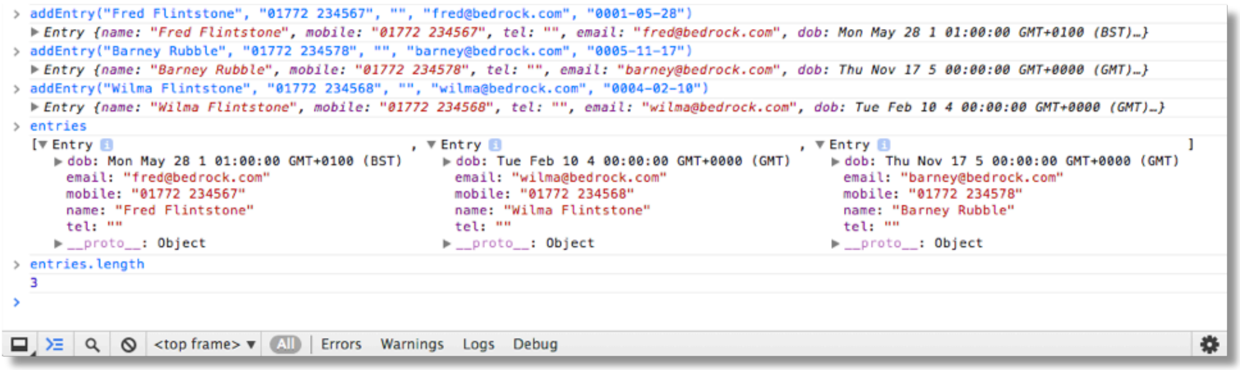
```
> e.changeName("James Marshall", "Hendrix")
  undefined
> e.displayName()
  "Hendrix, James"
> |
```

Figure 8.13: Having changed the name, is this the correct displayName()?

Note that this test reveals a slightly surprising result. I might have expected "Hendrix, James Marshall" to be the result when the new name was applied. However, because of how the displayable name is worked out (finding the last space to separate off the surname and using the first space to isolate the first name) we get an unexpected result. I'd say this should be fixed so that it was easier to differentiate between entries with the same first and last names – it would be quite normal practice to create entries for "John Smith" and "John Q Smith" so that they could be told apart on the screen. I'll leave that as an exercise for you to complete (hint – `lastIndexOf()` is still useful here).

Testing the Collection

Testing the entries collection is much like testing the Entry type, except that the results we get will inevitably be more complex and we'll need to try to think of all of the possible ways that it could go wrong. Tests 5-10 on page 84 is our basic checklist:

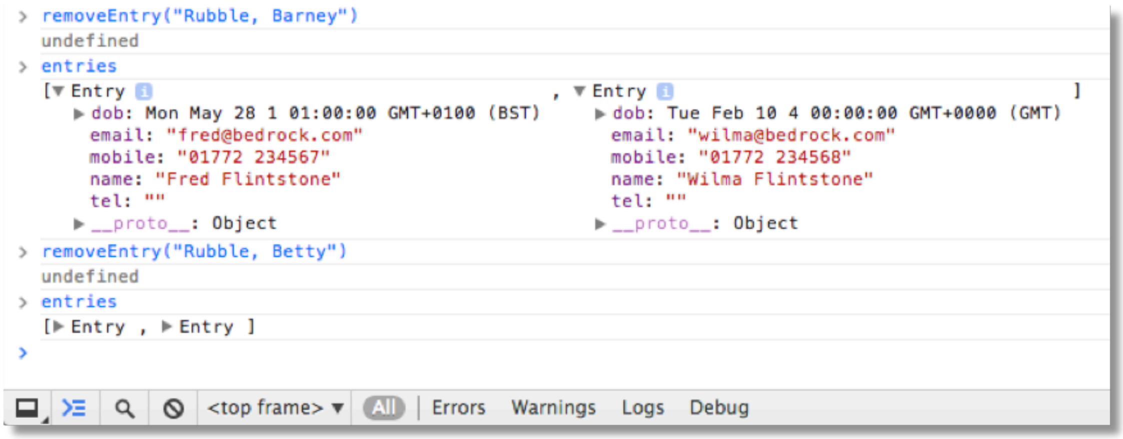


```
> addEntry("Fred Flintstone", "01772 234567", "", "fred@bedrock.com", "0001-05-28")
  ▶Entry {name: "Fred Flintstone", mobile: "01772 234567", tel: "", email: "fred@bedrock.com", dob: Mon May 28 1 01:00:00 GMT+0100 (BST)…}
> addEntry("Barney Rubble", "01772 234578", "", "barney@bedrock.com", "0005-11-17")
  ▶Entry {name: "Barney Rubble", mobile: "01772 234578", tel: "", email: "barney@bedrock.com", dob: Thu Nov 17 5 00:00:00 GMT+0000 (GMT)…}
> addEntry("Wilma Flintstone", "01772 234568", "", "wilma@bedrock.com", "0004-02-10")
  ▶Entry {name: "Wilma Flintstone", mobile: "01772 234568", tel: "", email: "wilma@bedrock.com", dob: Tue Feb 10 4 00:00:00 GMT+0000 (GMT)…}
> entries
  [▼Entry                                      , ▼Entry                                      , ▼Entry                                      ]
    ▶dob: Mon May 28 1 01:00:00 GMT+0100 (BST)    ▶dob: Tue Feb 10 4 00:00:00 GMT+0000 (GMT)    ▶dob: Thu Nov 17 5 00:00:00 GMT+0000 (GMT)
     email: "fred@bedrock.com"                      email: "wilma@bedrock.com"                     email: "barney@bedrock.com"
     mobile: "01772 234567"                         mobile: "01772 234568"                         mobile: "01772 234578"
     name: "Fred Flintstone"                        name: "Wilma Flintstone"                       name: "Barney Rubble"
     tel: ""                                        tel: ""                                        tel: ""
    ▶__proto__: Object                            ▶__proto__: Object                            ▶__proto__: Object
> entries.length
  3
>
```

Figure 8.14: Adding several entries to the collection

The first test (or set of tests) answers question 6 – can we add new entries to the list. Note that it is sensible to make sure that you can add more than one, and that having added a few, the `entries` collection shows up nicely in the console window, and has the correct `length` of 3. Note, I've omitted the telephone numbers from the entered data by using an empty string (only sensible, there were no landlines in the stone age).

Removing an entry is a similarly simple test:

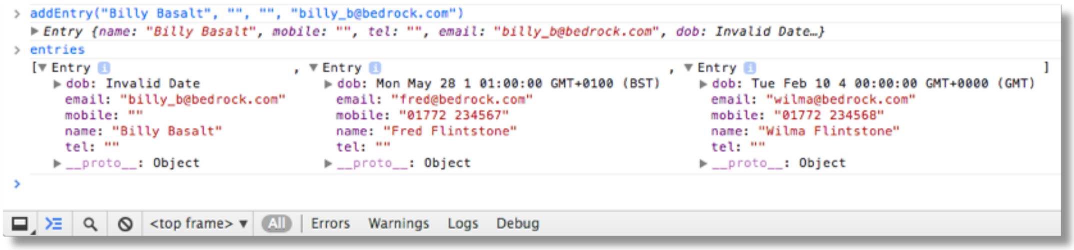


```
> removeEntry("Rubble, Barney")
  undefined
> entries
  [▼Entry                                      , ▼Entry                                      ]
    ▶dob: Mon May 28 1 01:00:00 GMT+0100 (BST)    ▶dob: Tue Feb 10 4 00:00:00 GMT+0000 (GMT)
     email: "fred@bedrock.com"                      email: "wilma@bedrock.com"
     mobile: "01772 234567"                         mobile: "01772 234568"
     name: "Fred Flintstone"                        name: "Wilma Flintstone"
     tel: ""                                        tel: ""
    ▶__proto__: Object                            ▶__proto__: Object
> removeEntry("Rubble, Betty")
  undefined
> entries
  [▶Entry , ▶Entry ]
>
```

Figure 8.15: Removing entries from the array

Note that I've done two tests – one to remove an entry that exists, and a second to remove an entry ("Betty Rubble") that was never added to the list.

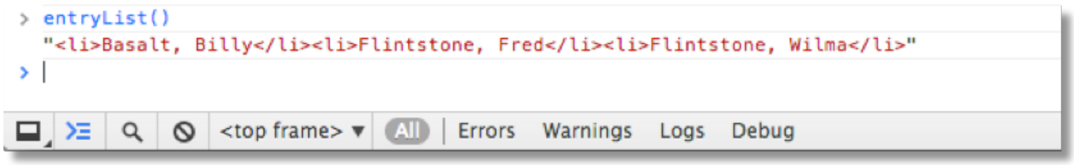
Before I can test the sortEntries() function, I'll need to add another entry since currently the list is already in alphabetical order (Fred, then Wilma). Ideally, I should add a record for someone who would appear as first or second in the list, since otherwise we don't know if sorting is taking place. Note that because sorting is done every time a new entry is added, all I need to do is add the new entry and then check its order:



```
> addEntry("Billy Basalt", "", "", "billy_b@bedrock.com")
  ▶ Entry {name: "Billy Basalt", mobile: "", tel: "", email: "billy_b@bedrock.com", dob: Invalid Date…}
> entries
  [▼ Entry                              , ▼ Entry                                      , ▼ Entry                                       ]
     ▶ dob: Invalid Date                   ▶ dob: Mon May 28 1 01:00:00 GMT+0100 (BST)    ▶ dob: Tue Feb 10 4 00:00:00 GMT+0000 (GMT)
       email: "billy_b@bedrock.com"          email: "fred@bedrock.com"                      email: "wilma@bedrock.com"
       mobile: ""                            mobile: "01772 234567"                         mobile: "01772 234568"
       name: "Billy Basalt"                  name: "Fred Flintstone"                        name: "Wilma Flintstone"
       tel: ""                               tel: ""                                        tel: ""
     ▶ __proto__: Object                   ▶ __proto__: Object                            ▶ __proto__: Object
>
```

Figure 8.16: A test designed to indicate whether entries are being sorted

Note that the entry for "Billy Basalt" (I'm sorry, but having started with a theme…) is correctly positioned as the first of the list. We have one remaining test:



```
> entryList()
  "<li>Basalt, Billy</li><li>Flintstone, Fred</li><li>Flintstone, Wilma</li>"
> |
```

Figure 8.17: The `entryList`() result, ready to insert into a `<ul>` element on a page

Having now tested (I'll stop short of saying "fully" tested) the code for the new type and the collection, we're ready to start building this into the existing user-interface – i.e. the HTML pages.

Integrating the HTML and Javascript

It is worth bearing in mind that a fair bit of effort has been put into making sure that the Javascript code that defines and manages our list of address book entries is fairly compatible with the HTML mark-up for the pages they will be displayed and edited on. Each entry has a displayName() method that will return the name in exactly the form we will use to display it. The collection will always be sorted (due to the way addEntry() was coded) and can return a simplified view of the collection (entryList()) that is marked-up ready to be dropped into the <ul> element on the main page.

We could go further in this respect; for example, we could try to integrate the Entry type with the mark-up on the 'entry' page (each entry could return its own <li> element). However, it is generally a good idea not to couple two separate bits of software too closely. If we make the HTML mark-up too dependent on the Entry type, or vice-versa, we're likely to create code that is inflexible and difficult to maintain.

The order of operations for this part of the job should be to start with general aims and move to more specific ones. So:

1. Add code so that the entries collection is displayed on the main page. We can test this by adding a couple of entries in the console and then executing the function that displays them
2. Add code to add a new entry from the user-interface
3. Add code so that when a user taps on an entry, it is displayed on the entry page

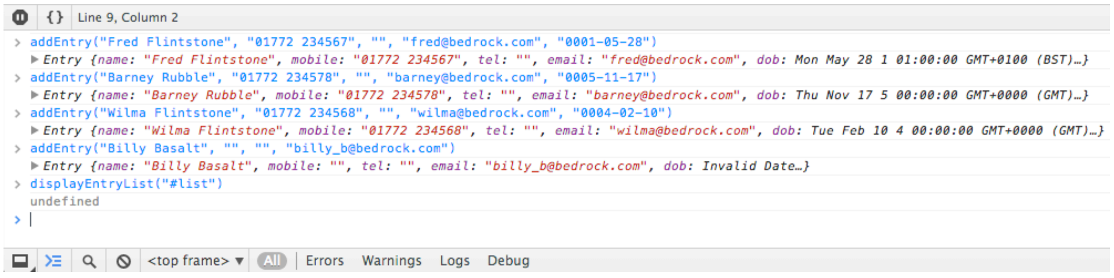
4. Add code so that when the user taps the Update button, any changes are reflected properly in the collection
5. Add code to delete an existing entry from the user-interface
6. Add code to save the list of entries to local storage whenever a change is made
7. Add code to retrieve the list of entries from local storage
8. Add start-up code so that the application automatically loads the list of entries if one exists and displays it

So, working through these:

```
function displayEntryList(listElement){
    $(listElement).html(entryList).listview('refresh');
}
```

Listing 8.10: A function to insert the list of entries into the display

The function to display a list of entries is refreshingly small and simple (largely due to the flexibility of jQuery). We pass the function the id for the list element (in the form `"#list"`), a jQuery selector is used to pick out this element from the DOM (the `<ul>`) and assign now HTML mark-up inside it (the `html()` method). Finally, this element is refreshed using the `listview('refresh')` method – note that jQuery always organises element methods so that they return the element that the method was called on, so we can simply follow the `html()` call with the `listview()` call immediately without using another selector statement. Because of the way the code is structured, we can call this function in the console and see its immediate result on the app's display:



```
Line 9, Column 2
> addEntry("Fred Flintstone", "01772 234567", "", "fred@bedrock.com", "0001-05-28")
  ▶ Entry {name: "Fred Flintstone", mobile: "01772 234567", tel: "", email: "fred@bedrock.com", dob: Mon May 28 1 01:00:00 GMT+0100 (BST)…}
> addEntry("Barney Rubble", "01772 234578", "", "barney@bedrock.com", "0005-11-17")
  ▶ Entry {name: "Barney Rubble", mobile: "01772 234578", tel: "", email: "barney@bedrock.com", dob: Thu Nov 17 5 00:00:00 GMT+0000 (GMT)…}
> addEntry("Wilma Flintstone", "01772 234568", "", "wilma@bedrock.com", "0004-02-10")
  ▶ Entry {name: "Wilma Flintstone", mobile: "01772 234568", tel: "", email: "wilma@bedrock.com", dob: Tue Feb 10 4 00:00:00 GMT+0000 (GMT)…}
> addEntry("Billy Basalt", "", "", "billy_b@bedrock.com")
  ▶ Entry {name: "Billy Basalt", mobile: "", tel: "", email: "billy_b@bedrock.com", dob: Invalid Date…}
> displayEntryList("#list")
  undefined
> |
```

Figure 8.18: Testing displayEntryList()

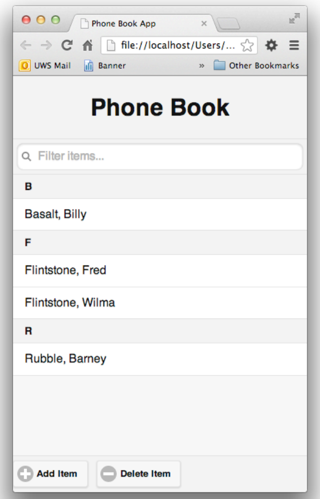


Figure 8.19: The result of calling displayEntryList()

Some points to note here:

- I did not re-type all of the `addEntry()` calls at the start of this test. You should recall that any statements that have already been been entered into the console can be re-executed by selecting with the up/down arrow keys and pressing Enter when you reach the one you want. You can even edit a statement before pressing Enter to use different data
- Because of the way the HTML content in the `<ul>` is replaced in the `displayEntryList()` function, the original list that was coded into the HTML is replaced. This is *a good thing*. It means we can always simply refresh the display with a call to `displayEntryList()` – i.e. if we had just *removed* or *altered* an entry, calling displayEntryList() takes care of updating the display automatically
- The call to `displayEntryList()` in the console window returned "undefined" as its result. If you check back, all of the other console windows in figures from 8.10 on show an "undefined" result when a function that returns nothing is called. That is fine, but maybe we should take a leaf out of the jQuery book, and always return something from a function – for example, `addEntry()` should return a reference to the Entry that was added, `removeEntry()` should return a reference to the one that was removed, `displayEntryList()` should return the list of `<li>` elements etc. Bear in mind I'm not going to make these changes here – this chapter is going to be long enough. However, you'll find the code on Moodle has been revised to work to this standard

Adding New Entries from the U-I

Adding a new entry is similar to the operation of adding a new item to the shopping list, except that now we don't need to bother about mark-up: simply add the new item and call `displayEntryList()`. The trick is in deciding where to put the code for this. We already have an `addEntry()` function that takes care of the fundamental operation; passing the data into a new `Entry` object and adding it to the list. What we could do with is a function that gets the appropriate values from the input fields on the second page and then calls addEntry(), passing these values to it as arguments. It turns out to be quite simple thanks to jQuery selectors:

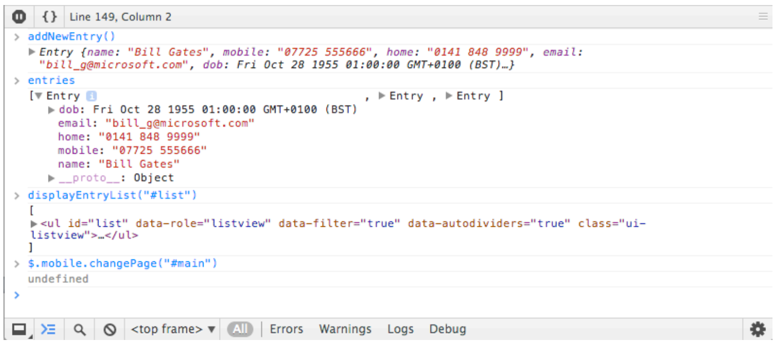
```
function addNewEntry(){
    var name = $("#fullname").val(),    // $("#<id>").val() gets the contents
        mobile = $("#mobile").val(),    // of a form field.
        tel = $("#tel").val(),
        email = $("#email").val(),
        dob = $("#bday").val();
    addEntry(name, mobile, tel, email, dob);
}
```

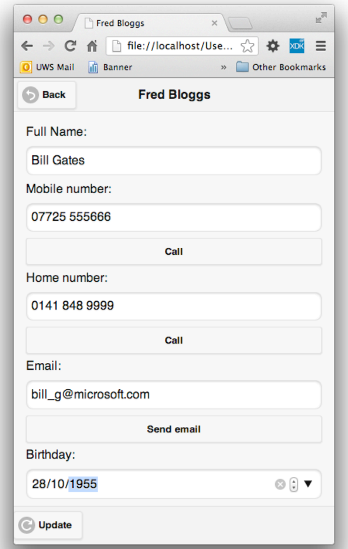
Listing 8.11: Adding a new entry

We can again test this function immediately using the Chrome Developer Tools; bear in mind it will be a bit more awkward because we're working between two different pages of the app to do it, but this test will give us some more confidence to develop add the final code doe the Add button. We'll need to do this in several steps:

1. Reload the App into the browser (so the new code is loaded)
2. Move to the "entry" page by adding "#entry" to the end of the app's URL. Note that the page heading will currently be "Fred Bloggs" since that is the dummy value placed in the mark-up
3. Fill in the various fields on the entry page (see figure 8.20)

4. Now, using the Developer Tools window (Ctrl+Shift+I), open the console view and enter the sequence of commands shown in figure 8.20. Note that the last command ($.mobile.changePage("#main") is a call to jQuery Mobile to move to a different page of the app (we want to se the result). Figure 8.21 shows the result of adding several entries like this
5. Repeat from step 2 to add other new entries


```
Line 149, Column 2
> addNewEntry()
  ▶Entry {name: "Bill Gates", mobile: "07725 555666", home: "0141 848 9999", email:
   "bill_g@microsoft.com", dob: Fri Oct 28 1955 01:00:00 GMT+0100 (BST)…}
> entries
  [▼Entry                                  , ▶Entry , ▶Entry ]
    ▶dob: Fri Oct 28 1955 01:00:00 GMT+0100 (BST)
     email: "bill_g@microsoft.com"
     home: "0141 848 9999"
     mobile: "07725 555666"
     name: "Bill Gates"
    ▶__proto__: Object
> displayEntryList("#list")
  [
  ▶<ul id="list" data-role="listview" data-filter="true" data-autodividers="true" class="ui-
  listview">…</ul>
  ]
> $.mobile.changePage("#main")
  undefined
>
```



Fred Bloggs

Full Name: Bill Gates

Mobile number: 07725 555666

Call

Home number: 0141 848 9999

Call

Email: bill_g@microsoft.com

Send email

Birthday: 28/10/1955

Update

Figure 8:20: Filling in the "entry" page and the console commands to use that data to add a new Entry object

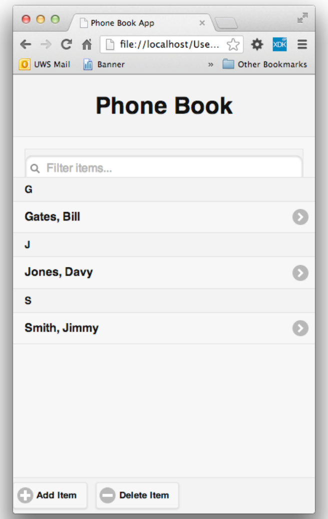


Figure 8.21: New entries resulting from following the process described above.

We now have confirmation that the code for adding and displaying entries is behaving as we would like it to. The final stage is to add corresponding code to the "Add" and "Update" buttons to make this work from the user-interface. We need to place these event handlers inside a `$(document).ready()` function call so that they are available *after* the HTML page is fully loaded. Note that we will want to use the Update button to update an existing entry as well as a new one, which means we will need to be able to tell whether the user pressed the Add button before the Update one (in which case it will be a new entry). Otherwise, we will need to apply the changes to the existing entry that was selected:

```
var currentEntry = "";     // This will stash the displayName() of a selected
                           // entry.  If it is "", it is a new entry.
$(document).ready() {

    $("#add").click(function() {
        currentEntry = “”;
        var e = new Entry();    // An empty one.
```

```
            displayEntry(e);        // Still to be defined.
        });

        $("#update").click(function() {
            if(currentEntry === “”){
                addNewEntry();
            }
            saveList();              // Still to be defined.
        });
    }
```

Listing 8.12: Code to add a new entry to the list

Note that in addition to the two event handlers (the first of which displays the "entry" page with an empty Entry object, the second takes the data from the "entry" page and uses it to create or update an Entry object), I've added calls to two other new functions. `saveList()` does the job of saving the entire collection to local storage - . `displayEntry()` takes an `Entry` object and puts its data on to the "entry" form.

```
    function displayEntry(e){
        $("#fullname").val(e.name);        // Note – .val() accesses or updates
        $("#mobile").val(e.mobile);        // any type of <input> field.
        $("#email").val(e.email);
        $("#home").val(e.home);
        $("#bday").val(e.dob);
        $("#name").text(e.name);           // Note, this is just an <h1> element
    }
    function saveList(){
        var strList = JSON.stringify(entries);    // See notes below.
        localStorage.phoneBook = strList;
    }
```

Listing 8.13: Functions for displaying an Entry object, and saving all of them

JavaScript Object Notation (JSON)

The `saveList()` function needs some explanation. We can use `localStorage` to access long-term storage on the device on browser the app runs in. However, `localStorage` will only work with strings and we are trying to store an array of `Entry` objects. JSON (JavaScript Object Notation) is an object that can take a data structure of any complexity in Javascript and turn it into a string in a standardised format. When we want to retrieve the collection of Entry objects, we can get the string back from `localStorage` and use `JSON.parse()` to recreate the original data. There is a bit more to it than that because we will need to recreate the whole Entry objects, all of which are based on a prototype that associates functions (e.g. `displayName()`, `isBirthday()`) with them. That is discussed below.

Back in the app, the final stage is to make sure that the list of entries is loaded and displayed when the app is loaded into a browser. The two function calls to do this should be at the top of `$(document).ready()` so that they are the first things to happen when the app starts:

```
    $(document).ready(function() {
        loadList();
        displayEntryList("#list");
        // ... the remainder of $(document).ready() ...
    });
```

Listing 8.14: When the app starts, we need to load and display the entries

The `loadList()` function needs to reverse what saveList() has done – i.e. it should recreate an array of Entry objects from the JSON text stored in localStorage:

```
function loadList(){
    var strList;
    strList = localStorage.phoneBook;     // Get the JSON data
    if(strList){                          // If there were entries...
        entries = JSON.parse(strList);    // ..use JSON to re-create them
        var proto = new Entry();          // This creates a prototype...
        for(e in entries){                // ...that we assign to all entries
            entries[e].__proto__ = proto;
        }
    } else {
        entries = [];
    }
}
```

Listing 8:15: loadList() has to re-create whole objects from the string data

Updating Existing Entries

Updating an existing entry is *almost the same process* as adding a new one; the only difference is that the data entered in the form should be packed back into an existing object instead of being used to create a new one. Note that the code attached to the Update button has already been sneakily arranged so that, currently, we add a new Entry only if the `currentEntry` variable is an empty string. We can add an `else` clause to that `if()` statement to say what happens if `currentEntry` is NOT empty:

```
$("#update").click(function() {
    if(currentEntry === ""){
        addNewEntry();
    } else {
        updateEntry();
    }
    displayEntryList("#list");
    saveList();
});
```

Listing 8.16: The full Update button process

In listing 8.16, it is now arranged that when the Update button is pressed we either add a new Entry or update an existing one, depending on the value stored in the `currentEntry` variable. Either way, we then display the entry list (which may have been changed) and save all entries.

The code to add a new entry is unchanged from listing 8.11. Updating an entry from data on the form is a matter of getting the current Entry object and re-populating it with data from the form:

```
function updateEntry(){
    var e = getEntryFromDisplayName(currentEntry);
    e.name = $("#fullname").val();
    e.mobile = $("#mobile").val();
    e.home = $("#home").val();
    e.email = $("#email").val();
    e.dob = $("#bday").val();
}
```

Listing 8.17: Changing data fields in an Entry object

Listing 8.17 uses a function, `getEntryFromDisplayName()`, to retrieve an Entry object based on the name that is displayed on the app's main page. This is done by stepping through the list of entries until a matching one is found:

```
function getEntryFromDisplayName(displayName){
    var index, e;
    for(index = 0; index < entries.length; index += 1){
        if(entries[index].displayName() === displayName){
            return entries[index];
        }
    }
    return null; // This should never happen
}
```

Listing 8.18: Looking up an Entry in the list

Note that there is a very optimistic comment in listing 8.18 ("This should never happen"). In the general scheme of things, that should be interpreted that it is 'very unlikely' to happen – even so, we shouldn't ignore the possibility. If we ever get a `null` value returned from this function, that will indicate that there is something we need to look into.

We're almost there. What remains is that we need to display an existing entry if the user clicks on one. There are two steps to this – if the user clicks on an entry, we need to detect which entry it was, which listing 8.18 takes care of. We then need to display that Entry on the entry page. Two top-level functions will take care of this:

```
$(document).on('click', "#list a", function() {
    // Get the text that the user clicked on. The selector (above) indicates
    // that it will be the text in an anchor element (<a>) inside the list of
    // names...
    currentEntry = $(this).text();
    // Get a reference (e) to the corresponding entry...
    var e = getEntryFromDisplayName(currentEntry);
    // Display it...
    displayEntry(e);
});

function displayEntry(e){
    $("#fullname").val(e.name);
    $("#mobile").val(e.mobile);
    $("#email").val(e.email);
    $("#home").val(e.home);
    // This is a bit of a beast, to do with the way the HTML <input> date type
    // expects dates to be formatted (toISOString() returns date + time).  We
    // need to isolate the date, which is the first 10 characters...
    $("#bday").val(e.dob.toISOString().substring(0, 10));
    // This simple sets the text in the page heading...
    $("#name").text(e.name);
}
```

Listing 8.19: Functions to handle an Entry being selected on the main screen, and to

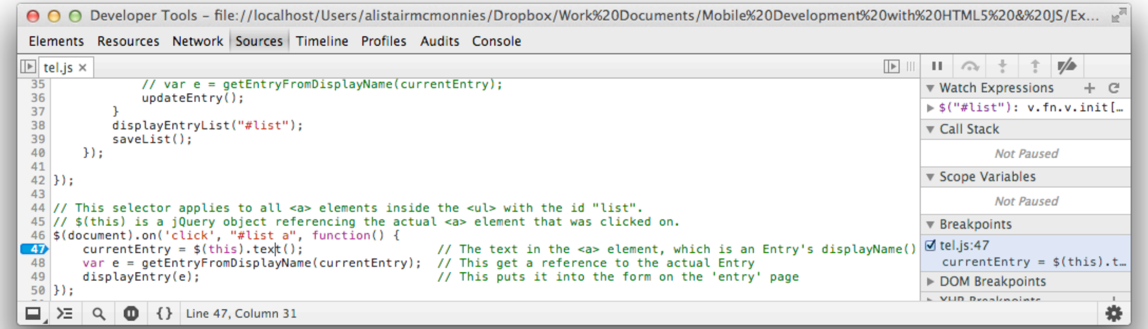
display that Entry on the entry screen

Final State Testing and De-bugging

At this stage, tests will involve using the app and checking that it does what we intended. However if anything did not behave as expected, we would again need to involve the developer tools in the browser to identify the problem.

Let's assume that when a name is clicked or tapped on the main page, the app does not respond by showing the corresponding Entry. How could we locate the source of the problem? (for the present code there is no such problem, but code that is being 'tweaked' is fragile, so it would be easy to get to that state. A good process to follow (for this app or one with a similar structure) is outlined below:

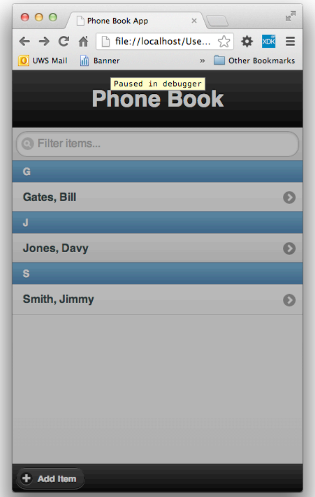
1. Open the app in a browser with Developer Tools (for example, Chrome or Firefox with Firebug installed)
2. Open the developer tools, select the 'Sources' view and locate the event handler code for the user clicking or tapping on a name in the list of names. Click in the grey margin to the left of the first line inside the event handler to set a breakpoint:

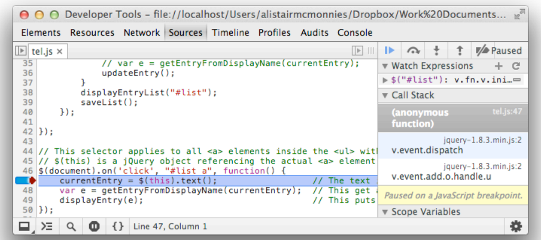

Developer Tools – file://localhost/Users/alistairmcmonnies/Dropbox/Work%20Documents/Mobile%20Development%20with%20HTML5%20&%20JS/Ex...
Elements Resources Network Sources Timeline Profiles Audits Console
tel.js
Watch Expressions
$("#list"): v.fn.v.init[...
Call Stack
Not Paused
Scope Variables
Not Paused
Breakpoints
tel.js:47
currentEntry = $(this).t...
DOM Breakpoints
Line 47, Column 31

```
35                // var e = getEntryFromDisplayName(currentEntry);
36                updateEntry();
37            }
38            displayEntryList("#list");
39            saveList();
40        });
41
42 });
43
44 // This selector applies to all <a> elements inside the <ul> with the id "list".
45 // $(this) is a jQuery object referencing the actual <a> element that was clicked on.
46 $(document).on('click', "#list a", function() {
47     currentEntry = $(this).text();                        // The text in the <a> element, which is an Entry's displayName()
48     var e = getEntryFromDisplayName(currentEntry);  // This get a reference to the actual Entry
49     displayEntry(e);                                      // This puts it into the form on the 'entry' page
50 });
```

Note the breakpoint indicator in the left margin at line number 47, and also in the list of breakpoints on the right

3. Now click on a name on the main page. If there are no current entries, add a few (at least more than one) so that you can check that the correct one is selected...



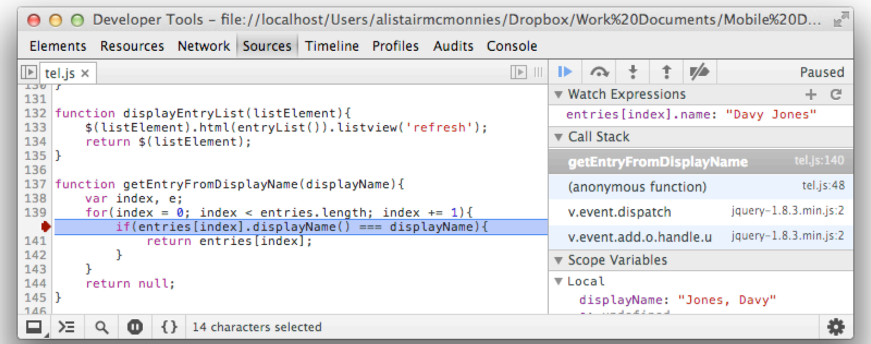

Developer Tools – file://localhost/Users/alistairmcmonnies/Dropbox/Work%20Documents...
Elements Resources Network Sources Timeline Profiles Audits Console
tel.js
Paused
Watch Expressions
$("#list"): v.fn.v.ini...
Call Stack
(anonymous function) tel.js:47
jquery-1.8.3.min.js:2 v.event.dispatch
jquery-1.8.3.min.js:2 v.event.add.o.handle.u
Paused on a JavaScript breakpoint.
Scope Variables
Line 47, Column 1

```
35                // var e = getEntryFromDisplayName(currentEntry);
36                updateEntry();
37            }
38            displayEntryList("#list");
39            saveList();
40        });
41
42 });
43
44 // This selector applies to all <a> elements inside the <ul> wit
45 // $(this) is a jQuery object referencing the actual <a> element
46 $(document).on('click', "#list a", function() {
47     currentEntry = $(this).text();                    // The text
48     var e = getEntryFromDisplayName(currentEntry);  // This get
49     displayEntry(e);                                  // This puts
50 });
```

Note that the browser screen showing the app should indicate "Paused in debugger" and the line with the breakpoint in the source view of the developer tools should be highlighted in blue.

4. We can now 'step' through the code using the controls at the top of the right-hand pane in the developer tools. will cause the current line (with the blue bar) to be executed and the code to stop on the next line. You can now hover the mouse cursor over the 'currentEntry'

variable name to inspect its contents:



Developer Tools – file://localhost/Users/alistairmcmonnies/Dropbox/Work%20Documents/Mobile%20D...

```
function displayEntryList(listElement){
    $(listElement).html(entryList()).listview('refresh');
    return $(listElement);
}

function getEntryFromDisplayName(displayName){
    var index, e;
    for(index = 0; index < entries.length; index += 1){
        if(entries[index].displayName() === displayName){
            return entries[index];
        }
    }
    return null;
}
```

```
Watch Expressions
  entries[index].name: "Davy Jones"
Call Stack
  getEntryFromDisplayName      tel.js:140
  (anonymous function)         tel.js:48
  v.event.dispatch             jquery-1.8.3.min.js:2
  v.event.add.o.handle.u       jquery-1.8.3.min.js:2
Scope Variables
  Local
    displayName: "Jones, Davy"
```

Elements Resources Network Sources Timeline Profiles Audits Console
tel.js
Paused
14 characters selected

Note that in my case, it shows "Jones, Davy", which was the name I clicked on. Note also that the next statement is the call to `getEntryFromDisplayName()`. I can 'step inside' that function by pressing the button in the toolbar.

Step into the `getEntryFromDisplayName()` function and use the button to step on to the inside of the for loop. You would find it useful to add a ***Watch Expression*** on the right-hand pane – click on the '+' symbol next to 'Watch Expressions' and enter the text `entries[index].name`. Then as you step through the code you will see the value changing as the name property of each entry is accessed.

If the `getEntryFromDisplayName()` function is working properly (it should be), the if() statement inside the for() loop will identify when the `displayName()` from the selected `Entry` matches the function's `displayName` parameter. When this happens, we can conclude that the function is behaving and step out of that function and on into the `displayEntry()` function, which will put the details of that Entry into the form on the second page.

We can continue like this, setting breakpoints at the start of a process, stepping through the code and inspecting variable and object values until we find unexpected results – usually a good indicator of a bug.

You'll find a full working version of the Phone Book code in Moodle in the Examples and Demos section at the end. There have been several small changes to make the app a bit more polished and to take care of scenarios I've not touched on here (e.g. what happens when you press the Call or Email button on the Entry page). A full version you can install on your phone is at this URL: http://goo.gl/IaKmP5

Debugging code is 95% craft (i.e. you can learn it and learn to use the tools that support it) and 5% instinct (which can be developed). Generally, the more code you examine in this way, the more finely honed your instincts will become. A good start would be to work through lab 1 and then try out the following web tutorials:

Debugging Javascript with Chrome: http://odetocode.com/blogs/scott/archive/2012/03/15/debugging-javascript-with-chrome.aspx

Javascript Debugging for Beginners: http://www.netmagazine.com/tutorials/javascript-debugging-beginners

Chrome Extensions – Tutorial: Debugging: https://developer.chrome.com/extensions/tut_debugging.html

Chrome DevTools - Debugging Javascript: https://developers.google.com/chrome-developer-tools/docs/javascript-debugging

Chrome DevTools – Using the Console: https://developers.google.com/chrome-developer-tools/docs/console

Questions

1. At the start of this chapter there are definitions of the terms Testing and Debugging. Characterise each of the following as one or other of these activities:
    a. The header of a web page appears in the wrong font; you need to find where this is happening in the code
    b. Having just added a new record to an Address Book, you need to verify that the total number of records is as expected
    c. When a record's email address is updated, the change has not taken place the next time you check. You need to find out whether this *always* happens, or only under certain circumstances
    d. You need to demonstrate that a new app installs properly on iPhones, 3 different types of Android phone and a Windows Mobile device
    e. A formatting error in address records could be due to a problem in Javascript code, HTML mark-up or CSS style sheets – you need to find out which so that appropriate person is sacked

Exercises

1. Get the Phone Book app working in the Chrome browser (download it from section 13 of the Moodle site). Using the Chrome debugging tools, find out where in the code the buttons that initiate a phone call or an email message get assigned the correct number/address information. To do this, you will need to locate the place in the code that displays an Entry when the user taps on it, so the point to start from is the event handler that responds to clicks on individual entries in the Javascript code.
2. Starting with a very simple jQuery Mobile app with one page which contains an empty <ul> listview, write a $(document).ready() event handler that installs the following list of Javascript objects <li> elements (note that each <li> will enclose a clickable link):

```
var objectList = [
{ title: "Debugging Javascript with Chrome",
  url: http://odetocode.com/blogs/scott/archive/2012/03/15/debugging-javascript-with-
chrome.aspx },
{ title: "Javascript Debugging for Beginners",
  url: "http://www.netmagazine.com/tutorials/javascript-debugging-beginners"},
{ title: "Chrome Extensions – Tutorial: Debugging",
  url: "https://developers.google.com/chrome-developer-tools/docs/javascript-debugging"}
];
objectList.prototype.getListItem = function() {
    return "<li><a href='" + this.url +  "'>" + this.title + "</a></li>";
}
```