HTML5 & Javascript: Lab 2

Objectives of this lab session:

- Create a constructor for a new object type
- Add methods to the object's prototype
- Create a simple (prototype) user-interface to work with the new type

Resources required:

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

Part 1: Using Constructor functions

In Javascript, the easiest way to create an object is to pack all of the object components into an object variable using a name:value syntax. However, this approach does not encourage consistency among a group of objects, so a constructor function is a preferable approach. In this lab, you'll use a constructor function to create a collection of objects.

Step 1: Starting the Project

To get going, you will need to create a new WebStorm project. This project will need two files (for now) – an html host file and a Javascript code file. The project will eventually be a simple (initially) appointments diary and/or to-do list.

1. Start up WebStorm and add a new project, called Appointments
2. Add a new HTML file (use the HTML5 template) and call it appointments.html
3. Add a new Javascript file and call it appointments.js.
4. We will be testing this project as we go along, so our first requirement will be to make sure the constructor function is available immediately the page has loaded. Add a script element to the html file (see lab 1 for details of this).

Our project will be expected to handle multiple appointments over a period of time, so rather than create a literal object we'll use a constructor. This will ensure that all appointments will have the same set of properties and methods:

5. Add the following constructor definition to the code file:

```
var Appointment = function(subject, description, date, time){
   this.subject = subject;
   this.description = description;
   this.datetime = new Date(date + " " + time);
   this.completed = false;
};
```

Note that the appointment date and time are combined into a single ***datetime*** property instead of individual date and time variables. This will simplify most of the date and time processing.
We'll expect an appointment to be capable of various behaviours – it will need to know when it is due, it should be capable of formatting itself for display and it should be able to indicate

how long until it is due (or by how much time it is overdue). Some prototype functions can be added to do this. Recall that a prototype function is one that will apply to every object created by an Appointment constructor:

```
Appointment.prototype.isDue = function(){
    var now = new Date();
    if(now >= this.datetime){                       // Note – a simple comparison
        return true;                   // with the current date and
    } else {                                        // time.
        return false;
    }
};

Appointment.prototype.whenDue = function(){
    return this.datetime – new Date(); // milliSeconds
}

Appointment.prototype.toString = function(){
   var s = this.subject + '\n' + this.description + '\n' +
      this.datetime.toString() + '\n';
   if(this.completed){
      s += "Completed\n\n";
   } else {
      s += "Not Completed\n\n";
   }
   return s;
};
```

Most of the above code should be clear enough, but the toString() function takes a little explaining. In Javascript (and other C-like languages), '\n' is encoding for a *single* character, which is a newline. Since a newline does not look like anything on a page, this coding is needed. The toString() method will return a multi-line string containing the appointment details. It won't be a very useful method for the final app, but will be useful for debugging.

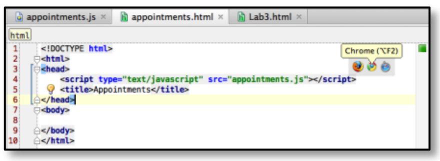
6. Our application will manage a collection of appointments. The quickest way to create a collection is to use an array (more details on this next week). Add the following statement directly below the Appointment prototype functions:

```
var appointments = [];
```

This statement creates an empty array (list) of Javascript objects. It can hold any type of object, but obviously we'll be using it for Appointments.

At this stage we can have a look at how our new type behaves, using the Chrome debugger to create and manipulate appointment objects.

7. Open the web application in Chrome (the quickest way is to go to the HTML page in WebStorm and move the mouse pointer to the top-right of the page. When browser icons appear, click on the Chrome one. Alternatively, select View→Web Preview from the menus:

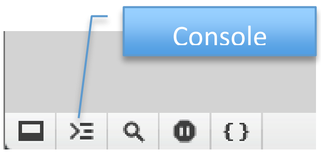


```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript" src="appointments.js"></script>
    <title>Appointments</title>
</head>
<body>

</body>
</html>
```

When the browser appears it should show an empty page.

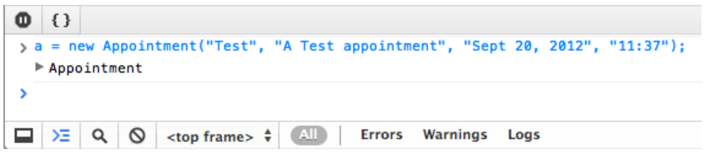
8. Open up the Chrome Developer tools (View→Developer→Developer Tools) and click on the Console button to open the console pane. Since the html page has been loaded, any script available to the page is also available to the Console.



9. Enter the following statement (substituting appropriate values) to create a new appointment. ***Make sure the date entered is today's date*** (in the correct format as shown) ***and the time is a couple of minutes from now*** (add two minutes to the time shown on the Windows task-bar):

   a = new Appointment("Test", "A Test appointment", "Sept 20, 2012", "11:37");

   If your code has been entered correctly, the console window should respond by showing an Appointment object below your line of code:



```
> a = new Appointment("Test", "A Test appointment", "Sept 20, 2012", "11:37");
  ▶ Appointment
>
```

   You can click on the small arrow to the left of the name Appointment to open up the object and show its data members (completed, datetime, subject & description), plus __proto__, which is the variable that refers to the object's prototype. Open up __proto__ to view the list of prototype functions.

10. Provided you've done all this fairly quickly, if you enter a.isDue() in the console it should show a result of false. If you took too long over this and went past the appointment time, simply press the up arrow to review the statements you've already entered, edit the appointment time and try again. If you now wait until the time reaches the time you entered when you created the appointment and repeat this it should show true.

11. Enter a.toString() to see the result of the toString() function. Note that this is fairly useless for displaying the appointment on a web page, where other formatting rules apply. However, it can be useful to view objects in the console in an easy to view format, so writing a toString() function for a new type is usually quite helpful.

12. Finally, enter a.whenDue() to see the number of milliSeconds until (or since – a negative value) the appointment is (was) due.

Exercises

Add new prototype functions to the appointment type to provide the following features. In each case, test the new function in the Console window:

a) howManyDaysTill() should return the number of days until an appointment (based on taking the integer part of milliseconds divided by the number of milliseconds in a day).

b) howManyHoursTill() should return the number of hours till the appointment is due

c) getDate() should return only the date part of the datetime element. A Javascript date has the toDateString() function that will return this information

d) getTime() should return only the time of day part of the datetime element. A Javascript date object has the getTime() function, but this returns extraneous data that would not look good on a web page (try entering a.datetime.getTime() in the console to see). A better format can be cooked up by returning the results of getHours() and getMinutes()

with a colon ':' between them

e) tableRow() should return an appointment formatted as a row of a table. Note that a row in an html table is a sequence of HTML mark-up tags surrounding the data you want:

```
<tr>
    <td>First data element</td>
    <td>Second data element</td>
    … more data elements …
</tr>
```

These don't have to be formatted on separate lines. For example, if I just wanted to show the subject of an appointment in a row of a table, I could use:

```
"<tr><td>" + this.subject + "</td></tr>"
```

Ideally, your table row should show subject, due date, due time and whether the appointment was completed or not. Use your appointment getDate() and getTime() functions in this.

Adding a simple (too simple) User-Interface

Creating a full user-interface for an HTML application can be quite a big task, so for this week we'll concentrate on getting something simple to work to demonstrate what we have done already in the Appointment type. Lab 3 will concentrate on building a full user-interface.

Currently our HTML page is more or less blank:

```
<!DOCTYPE html>
<html>
<head>
   <script type="text/javascript" src="appointments.js"></script>
   <title>Appointments</title>
</head>
<body>

</body>
</html>
```

For our working prototype will need to be able to create new appointments and show all of the existing ones. Fortunately we've done much of the work for this already (in the tableRow() method of the Appointment type).

Creating a new appointment

For this, we'll need to put a button on the page, which is fairly simple.

1. Add the following html tag inside the <body> tag:

   ```
   <button id="new">New Appointment</button>
   ```

2. Save the html file and open the page in a browser to see the result of this – the blank page should now contain a button with the specified text (New Appointment).

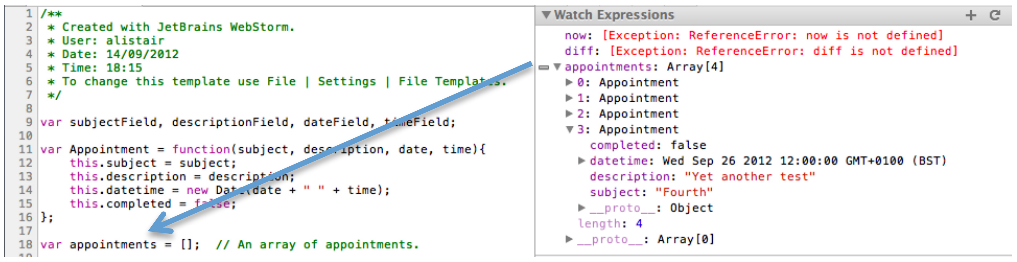
We now need to 'wire-up' this button to an event handler, so that we can run code to create a new appointment. Since we'll need the event handler to be available as soon as the page loads, the best place to put it is into the window.onload event.

3. Move to the Javascript file and add a new function below all of the other code:

```
window.onload = function(){
    var newButton = document.getElementById("new");
    newButton.onclick = function(){
        var subj = prompt("Enter a subject title for the appointment:");
        var desc = prompt("Enter a description of the appointment");
        var date = prompt("Enter the appointment date in the format (e.g.) 'Sep 25, 2012'");
        var time = prompt("Enter the appointment time in the format hh:mm");
        var a = new Appointment(subj, desc, date, time);
        appointments.push(a);
    };
};
```

4. Reload the page and press the New Appointment button. You will be prompted to enter the four pieces of information – you can enter anything as a subject and description (since strings can contain any text you can type), but you will need to be careful entering the date and time. You can enter a Javascript date as (for example) “September 25, 2012”, “Sept 25, 2012”, “Sep 25, 2012”, “2012-09-25” or “2012-9-25”. Anything outwith these formats may not be recognised as a date (this can be annoying, since our natural format is “25/09/2012”, but we’re stuck with it). Entering time is a bit easier – “12:25” is the same format most people would use

5. Move to (or re-open) the developer tools and select the Scripts page. Open the appointment.js script (unless it is already open) and right-click on the appointments array variable. Select Add to Watch from the context menu. You should now be able to inspect the appointment you added in the Watch window (on the right of the Developer Tools window at the top) – open up the appointments Watch entry and drill down to view the individual parts:



```
/**
 * Created with JetBrains WebStorm.
 * User: alistair
 * Date: 14/09/2012
 * Time: 18:15
 * To change this template use File | Settings | File Templates.
 */

var subjectField, descriptionField, dateField, timeField;

var Appointment = function(subject, description, date, time){
    this.subject = subject;
    this.description = description;
    this.datetime = new Date(date + " " + time);
    this.completed = false;
};

var appointments = [];  // An array of appointments.
```

```
▼ Watch Expressions
  now: [Exception: ReferenceError: now is not defined]
  diff: [Exception: ReferenceError: diff is not defined]
▼ appointments: Array[4]
  ▶ 0: Appointment
  ▶ 1: Appointment
  ▶ 2: Appointment
  ▼ 3: Appointment
      completed: false
    ▶ datetime: Wed Sep 26 2012 12:00:00 GMT+0100 (BST)
      description: "Yet another test"
      subject: "Fourth"
    ▶ __proto__: Object
    length: 4
  ▶ __proto__: Array[0]
```

Viewing Appointments

For this prototype, the last task is to let the user see appointments that have been added. Since each appointment is already able to generate and HTML table row of its own data, this won’t be a big task:

6. Add HTML elements to mark out where a table of appointments will be visible. A simple <div> element can be used as a container for a block of other html. Add it to the end of the <body> element, just below the button :

```
<div id="table">
</div>
```

7. Add a function to generate a table containing the html appointment data:

```
var showTable = function(){
    var tableDiv = document.getElementById("table"),
        table = "<table border='1'>" +
            "<thead><th>Date</th><th>Time</th><th>Subject</th><th>Completed</th></thead>";
    for(var i=0, j=appointments.length; i<j; i++){
        var appt = appointments[i];
        table += appt.tableRow();
```

```
      }
      table+="</table>";
      // Now add the table to the page...
      tableDiv.innerHTML = table;
    };
```

Note that I've had to split the statement assigning html tags to the table variable over a couple of lines – you can enter it on a single line and save having to join two strings.

8. Add a statement to the end of the newButton.onclick() function (inside window.onload()) to call this function:

   ```
   showTable();
   ```

9. Reload the browser and try adding a few appointments.

Exercises

1. Your appointments should show up as rows of a table, complete with headings. The Completed column should currently show as text – either "Completed" or "Not Completed". Instead, you could add a Checkbox style input element inside this table cell – the syntax of this is <input type='checkbox' value='…'/>. I've omitted the actual value here, since you would need to insert the value of the .completed property into this html. Do that by closing the quotes, adding this.completed and then adding the closing tag as a separate string – i.e.

   ```
   "<input type='checkbox' value=' " + this.completed + "/>"
   ```

   Amend the tableRow() method to display the Completed values as checkboxes.

2. You can add a timer element to Javascript code so that a function can be called after a given period, or at given intervals. The syntax for this is:

   ```
   setInterval( function(){
       // code to execute
   }, <interval>);  // <interval is a value in mS (60000 is once per minute).
   ```

   If you wanted the appointment timer to alert the user of due appointments, you could use this function to check the isDue() method of each appointment, and pop up an alert if any were due. In that case the function assigned to setInterval() would be something like:

   ```
   for(var i=0; i=appointments.length; i++){
       var appt = appointments[i];
       if(appt.isDue()){
          alert(appt.subject);
       }
   }
   ```

3. If you got part 2 working, expand this to alert the user of appointments due in 1 hour or less (i.e. using the howManyHoursTill() function).

End of Lab2