Programming the Internet—Lecture Notes

Week 4: JavaScript II

Functions

In Week 3, we looked at a simple function that provided a way of grouping a number of lines of code so that when we needed to use that code a number of times in a program, we simply 'called' the name of the function rather than typing out the code. Thus, we had the following function definition (in Figure 4: addNumbers.js):

```
function addNumbers() {

var numberOne =
parseInt(document.getElementById("firstNumber").value);
var numberTwo =
parseInt(document.getElementById("secondNumber").value);

var total = parseInt(numberOne + numberTwo);

alert ("The total is " + total);
}
```

Thereafter, to call this function and use its code, we would simply write *addNumbers()* in our JavaScript program. It is more usual for us to want to also pass information to a function and for it to return information to us. Last week, we were introduced to the *alert* method to which we passed information. In the above example we have:

```
alert ("The total is " + total);
```

Here, we were using one parameter or argument (the part in the brackets) to pass a string to the method (function) called *alert*. We can apply the same principle in defining our own functions. Let's revise our simple function so that instead of specifying the two numbers to be added within the function, we pass those numbers to the function. This will make it more flexible so that we are able to add any two numbers. Here is a revised function definition:

```
function addNumbers (firstNumber, secondNumber) {
var total;
total = parseInt(firstNumber + secondNumber);
}
```

In calling this function and passing two pieces of information to it, we should write this in our JavaScript program:

```
addNumbers (numberOne, numberTwo);
```

Let's now see what is going on here. In the function definition, *firstNumber* and *secondNumber* are referred to as *formal parameters*. They may be thought of as the variable names for our two numbers that will be used inside

the function. Once they are defined as formal parameters in the function header, they do not need to be declared again with a *var* declaration. We say that the 'scope' of these formal parameters lies within this function only; they cannot be used outside it. Note that we have also declared a variable called *total* within the function. This is known as a *local variable*, and it too cannot be used outside this function.

This is good programming practice. We can declare global variables that can be used anywhere in our program. However, when our programs become more complex, it is easy for the values in these global variables to be changed unexpectedly because all parts of the program have access to them. In order to counter this problem, we can define functions with local variables and allow information to be passed to them only by the use of parameters.

So why are the names of the variables different when we call the function in our main program? The variables *numberOne* and *numberTwo* are known as *actual parameters*. We here assume that they are variables that have been declared somewhere in the main program, possibly as global variables, and have been given values. They might well have received these values from an HTML form. To emphasise the fact that *firstNumber* and *secondNumber* can be used only within the function, we give our two numbers different variable names outside the function. Hopefully, this will become less confusing as you work through the example.

When we call the function with:

```
addNumbers (numberOne, numberTwo);
```

the JavaScript interpreter will look for a function called *addNumbers*, the definition of which indicates that it is expecting two parameters to be passed to it. On finding this function, it will then match the actual and formal parameters. Because it comes first, *numberOne* will be matched with *firstNumber* and the value stored in *numberOne* will be passed to *firstNumber*. Similarly, the value stored in *numberTwo* will be passed to *secondNumber.* We shall assume that the two numbers which we assigned to *numberOne* and to *numberTwo* are still 4 and 6, as we assigned them in Week 3. These values are now safely inside our function, ready to be processed. The line of code:

```
total = parseInt(firstNumber + secondNumber);
```

is now executed and *total* should contain the value 10. The total of the two numbers is now contained in a local variable that cannot be used outside the function. How do we then get the total back into the main program? We need an extra line of code:

```
function addNumbers (firstNumber, secondNumber) {
var total;
total = parseInt(firstNumber + secondNumber);
return total;
```

```
}
```

The *return* statement is what enables us to pass information back to the main program. This returned information is actually stored in the identifier (function name) used in the function call. In our example of a function, we could write the following:

```
var addTotal = addNumbers (numberOne, numberTwo);
```

Here, the function name *addNumbers* is not only passing information to the function, but it is receiving the total back from the statement *return total*. The function name is temporarily acting like a variable. This total is then being assigned to (stored in) the variable *addTotal* so that we can use it in the rest of our program. We have used two parameters *(numberOne* and *numberTwo*) in our example, but it could have been one or three or more.

It is recommended that any JavaScript functions be placed in an external JavaScript file.

Let's pull these ideas together with a complete coded example. Firstly, the HTML:

```
<!DOCTYPE html>
<html>
<head>
<title> Adding Two Numbers - Function Using Parameters </title>
<meta charset="utf-8" />
<script type = "text/javascript" src="addNumbersParams.js"></script>
</head>
<body>
<form>
<p>
<label for="firstNumber">Type an integer (a whole number).</label><br
/>
<input name = "firstNumber" id = "firstNumber" type = "text" size =
"6" maxlength = "8"  /><br />
<label for="secondNumber">Type another integer.
</label><br />
<input name = "secondNumber" id = "secondNumber" type = "text" size =
"6" maxlength = "8"  /><br />
</p>
<p>
<input type = "button" id = "addButton" value = "Add Numbers" onclick
= "RetrieveValuesAndAdd()" />
</p>
</form>
</body>
</html>
```

Figure 1: addNumbersParams.html

This code is very similar to an example we had last week that asks a user to input two numbers. Here is the JavaScript:

```
function RetrieveValuesAndAdd() {
var numberOne =
parseInt(document.getElementById("firstNumber").value);
var numberTwo =
parseInt(document.getElementById("secondNumber").value);
addNumbers (numberOne, numberTwo);
}

function addNumbers (firstNumber, secondNumber) {
var total;
total = parseInt(firstNumber + secondNumber);
alert ("The total is " + total);
return total;
}
```

Figure 2: addNumbersParams.js

For convenience and clarity we have separated the process of retrieving two numerical values from the HTML form and the process of adding the two numbers together. The parameterless function (it has no parameters) named *RetrieveValuesAndAdd* gets the numbers from the form and stores them in variables named *numberOne* and *numberTwo*. These variable names are used as parameters to call our new version of the *addNumbers* function:

```
addNumbers (numberOne, numberTwo);
```

The value in *numberOne* is passed to *firstNumber*, and the value in *numberTwo* is passed to *secondNumber* inside the *addNumbers* function. The two numbers are added together and stored in the variable *total*.

```
total = parseInt(firstNumber + secondNumber);
```

parseInt is used to ensure that the numbers are stored as integers rather than strings.

The JavaScript program finishes by using an *alert* box to output the total to the screen, and returning the value to the place from which *addNumbers* was called, which is in the *RetrieveValuesAndAdd* function. The *RetrieveValuesAndAdd* function could now do further things with these two values if it wished.

Arrays

Sometimes we want to store a number of items of information of the same type, perhaps a collection of surnames or student marks. If a variable can be thought of as a container for storing information, an *array* can be thought of as a set of containers. In an array, each container is given a number. In JavaScript, the numbering starts at 0. This number is known as the *index*. The first position in an array called *surnames* would be referred to as follows:

```
surnames[0]
```

The second position would be *surnames[1]* and so on. We should declare this array as follows:

```
var surnames = new Array(20);
```

This will declare an array which can contain 20 elements (it has 20 containers for storing information). The last element will be *surnames[19]*. Note that we use square brackets for referring to an array element but rounded brackets, or parentheses, for specifying the number of elements in an array declaration. A value can be assigned to an array element as follows:

```
surnames[0] = "Smith";
```

Mathematical operations can also be done on arrays. If *salaries* is an array of numbers, then the statement:

```
sum = salaries[2] + salaries[3];
```

will add the value of the third element of *salaries* to the fourth element (remember that the first element is *salaries[0]*) and store the total in *sum.*

The *for* loop

Some problems require us to perform a similar action a number of times. In programming, this is known as *iteration*. Programming languages such as JavaScript use loops to enable us to do this.

Let us suppose we have an array which will store student marks, expressed as a percentage. We could declare that array as follows:

```
var marks = new Array(25);
```

Suppose we should like to go through all the elements in that array, setting the initial value to 0. We should be said to be 'initialising' the array. It would be tedious to use 25 assignment statements to do this, so an alternative would be to use a *for* loop.

```
for (i = 0; i < marks.length; i++)
{
marks[i] = 0;
}
```

This example has three parameters.

- *i = 0:* The *for* loop is going to do something a number of times, and it will achieve this by keeping a count of how many times it has done it. We are going to use the variable *i* to keep that count. This first parameter is saying that we are going to start counting at 0 (because we are going to start by putting some value in position *marks[0]* of the array). Therefore, our counter is initially assigned the value 0.

- *i* < *marks.length*: If the *for* loop is going to do something a number of times, it needs to know when to stop. JavaScript helps us by automatically storing the number of array elements (the number of containers) in a variable *marks.length*. In our example, *marks.length* will contain the value 25. This parameter is saying the following: Keep going as long as the value of *i* is less than *marks.length* (less than 25). Thus, when *i* is 24 we are pointing at the 25th element in the array which we want to set to 0. When *i* is 25, we are pointing at a position beyond the end of the array and the loop stops.

- *i*++: The third parameter tells the program to increment *i* (add one to itself) every time we go through the loop. It is equivalent to writing *i = i + 1;*. This will enable us to do something with *marks[0]*, then increment *i,* and do something with *marks[1]* and so on.

Between the braces (curly brackets), we specify what it is that should be done each time we go through the *for* loop. In this case, we are setting the value stored in array position *marks[i]* to 0. In our program, the value of *i* will increase from 0 to 24 by increments of 1. Thus, by the time the *for* loop stops, every position in the array will have the value 0 assigned to it.

The *while* loop

Sometimes we shall not know how many times we want to do something. Instead, we may want to stop looping when some condition is met. The *while* loop helps us to achieve this. Here is an example which checks for incorrect values (assume that the array named *marks* has been filled with the percentage scores of a class of students):

```
i = 0;
while ((marks[i] >= 0) && (marks[i] <= 100) && (i < 25))
{
i++
}  // End of while loop
if i < 25 /* The loop has terminated before reaching the end of the
array */
{
alert ("We found an invalid mark at array position marks[" + i +
"]");
}
Else  // We reached the end of the array
{
alert ("We reached the end of the array and no invalid marks were
found.");
}
```

The loop is searching through the array of marks looking for any values that are less than 0 or greater than 100. These percentages will obviously be invalid. If an invalid mark is found, an alert will be placed on the screen, indicating the array position where the error has occurred.

The loop's condition is contained in the brackets after the word *while*. It is saying: While the marks we are looking at are not less than 0 and not greater than 100 and we have not reached the end of the array *(i < 25)*, then increment the value of *i (i++)* so that we can examine the next mark in the array.

A logical 'and' in JavaScript is denoted by '&&'. This is an example of Boolean logic. A logical 'and' statement is only correct when all of its parts are correct. So, if the current mark is less than 0 or greater than 100 or we have reached the end of the array, the loop will stop or exit. At that point an *if* statement will decide whether we reached the end of the array without finding an error or not. An appropriate alert message will be placed on the screen.

The logical 'or' is denoted by '||'. The following while condition:

```
while ((marks[i] >= 0) && (marks[i] <= 100) && (i < 25))
```

can also be written using 'or 'and the 'not' symbol (!) to achieve the same result. The following is logically equivalent to the 'and' version (they are true in exactly the same circumstances):

```
while (!(marks[i] < 0) || (marks[i] > 100) || (i >= 25))
```

It can read as follows: While it is not true that (the current mark in the array is less than 0) or (the current mark is greater than or equal to 100) or (*i* is greater than or equal to 25), then the loop will keep going.

Do while

The *while* loop tests to see if the condition is true at the beginning of the loop. In that case it is possible that it will never execute the code within the loop as it might fail the test at the first attempt. For cases where the programmer wants the code to execute at least once, it is possible to use a *do while* loop which carries out the test at the end of the loop. Our example can thus be rewritten:

```
i = -1;
do
{
i++
} while ((marks[i] >= 0) && (marks[i] <= 100) && (i < 25));
// End of do…while loop
if i < 25
{
alert ("We found an invalid mark at array position marks[" + i +
"]");
}
else
{
alert ("We reached the end of the array and no invalid marks were
found.");
}
```

Note that the initial value of *i* is set as -1 so that when the value is incremented at the beginning of the loop, i has the value of 0 (it is pointing at the first element in the array).

Validation and verification

JavaScript is often used to check user input before it is sent to a Web server. Examples of validation would be tests of whether a value is outside a permissible range (as in our example above), whether input contains impermissible characters and whether data is of the wrong type. Tests might try to rule out a month (expressed as a number) that is greater than 12, an e-mail address that does not contain an @ symbol and a percentage figure that contains alphabetical characters. Valid input may still be incorrect, but validation checks can pick up some of the more obvious impermissible input. Very sophisticated validation can be performed by using regular expressions, which provide a notation to define a template to which input (such as an e-mail address) should conform. For those interested, visit W3 Schools (2012).

Verification tries to identify cases where the user mistypes input but the input is permissible and hence is not detected by validation checks. A common form of verification is to ask users to type their password twice and then get JavaScript to check that the two versions are the same. This approach is particularly useful when a user is changing his or her password. A mistyping of the suggested new password could result in the user being locked out of his or her account.

In our discussion of the *while* loop, we have already seen how JavaScript can be used to check whether a mark is within a certain range (0-100). We shall now look at some more examples, which involve validation with an HTML form, employing *onsubmit* to make the checks before the form data is submitted to a Web server. Firstly, the HTML we shall use:

```
<!DOCTYPE html>
<html>
<head>
<title> A Form to Be Validated by JavaScript </title>
<meta charset="utf-8" />
<script type = "text/javascript"
src="JavaScriptValidation.js"></script>
</head>
<body>
<form onsubmit = "return validateJS()">
<!-- Putting return in front of the function name means that we are
expecting it to return a value. In this case, the value is either
true or false. If it returns true (meaning all fields have been
completed), the form will submit. If it returns false, the form will
not submit until the fields are filled in. -->
<p>
<label for="firstName">Your first name</label><br />
<input type = "text" name = "firstName" id = "firstName" size = "10"
maxlength = "15"  /><br />
```

```
<label for="surname">Your surname</label><br />
<input type = "text" name = "surname" id = "surname" size = "10"
maxlength = "15"  /><br />
</p>
<fieldset>
<legend> How many years have you been studying for your degree?
</legend>
<input type = "radio" name = "years" id = "Y5" value = "5" /> <label
for="Y5"> 5 </label> <br />
<input type = "radio" name = "years" id = "Y4" value = "4" /> <label
for="Y4"> 4 </label> <br />
<input type = "radio" name = "years" id = "Y3" value = "3" /> <label
for="Y3"> 3 </label> <br />
<input type = "radio" name = "years" id = "Y2" value = "2" /> <label
for="Y2"> 2 </label> <br />
<input type = "radio" name = "years" id = "Y1" value = "1" /> <label
for="Y1"> 1 </label> <br />
</fieldset>
<p>
<input type = "submit" name = "submit" value = "Submit" />
<input type = "reset" value = "Clear" />
</p>
</form>
</body>
</html>
```

Figure 3: JavaScriptValidation.html

This is a short HTML form that contains two text boxes and a set of radio buttons. It will look like this:

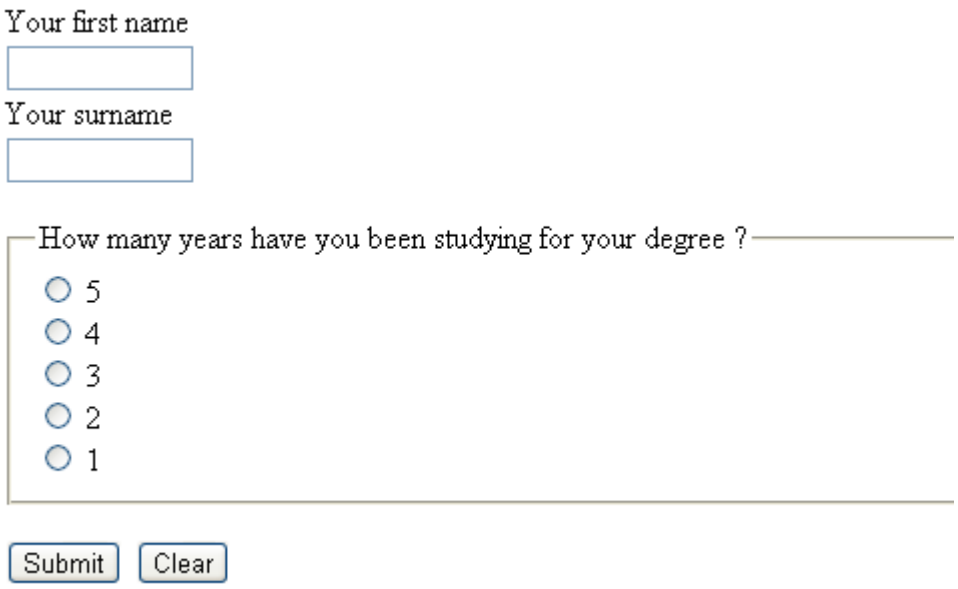

Figure 4: Output from JavaScriptValidation.html

When we first used a set of radio buttons we included a ‘Not Answered’ option which was checked when the HTML file opened. This warned us that the user had not completed this question. In our current example we are going to use

JavaScript to validate the form and alert us when no radio button is selected. For our current purposes we are going to assume that all fields are required, i.e. that the user must answer all questions.

We have seen that giving an HTML element an *id* helps us to manipulate it using *getElementById*. We could adopt this approach in validating the form by including lines of JavaScript code like this:

```
var firstname = document.getElementById("firstName").value;
var surname = document.getElementById("surname").value;
if ((firstname === "") || (surname === "")
/* If either of the text boxes has been left empty */
   {
   alert("You have not filled in a text box.");
   }
```

That will work where there are only two text boxes but, in a large form with many fields, it becomes a little messy. Fortunately, JavaScript has two other functions, *getElementsByTagName* and *getElementsByName,* which will be of use to us. JavaScript does, behind the scenes, group together collections of objects that work very much like arrays. There are, therefore, ways of accessing all the tags of a certain kind or attributes with the same name because they will have been bundled together for us by JavaScript.

Remember that in a set of radio buttons we normally give each button a separate *id*, but they all have the same *name*. It is because they have the same *name* that they form a set, and only one of the set can be selected at any time. All items with the *name* called *years* have been bundled together for us in a collection by JavaScript so that we can use *getElementsByName* to search through the radio buttons in our sample code to see whether any of them has been checked.

If you are observant, you will have noticed something slightly different in this line of code from Figure 3:

```
<form onsubmit = "return validateJS()">
```

Instead of just providing the name of a JavaScript function, we have used the word *return* in front of it. When the user clicks the *Submit* button, some JavaScript code will check to see if all the fields have been completed. Without the word *return*, the browser might then go on to submit the incomplete form data to a Web server.

By using *return* we are saying that only when the value ‘true’ is returned from the function *validateJS()* will the form be submitted to the Web server. If something is not completed, it will not submit the form. You will therefore see that, in the JavaScript code in Figure 5 below, there is a *return* statement at the end of the function *validateJS()* which will return the value true or false to the HTML form.

There are extensive comments in the validation code in Figure 5 which should explain how it works. If in doubt, ask your Instructor for clarification. It might well be easier to use a text editor to read the source code we have provided in the Code Examples zip file rather than viewing it in these Lecture Notes. This is because your text editor will not tend to wrap lines due to limited page width.

```
function validateJS() {
validateTextBoxes();
/* Execute the code in the function validateTextBoxes */
validateRadios();
/* Execute the code in the function validateRadios */
return (validateTextBoxes() && validateRadios());
/* If both validateTextBoxes and validateRadios have returned true
(meaning that all fields have been completed) then validateJS will
return true to the HTML file and the form will submit. If not, the
form will not submit until the user fills in the necessary fields.
*/
}

function validateTextBoxes()  {
var all_input = document.getElementsByTagName("input");
/* Get all the elements with the tag name "input" and put them in a
collection called all_input */
var completed = true;
/* The value of "completed" is set to true before the for loop begins
and will only be set to false if one of the text boxes is not
completed */
for (i = 0; i < all_input.length; i++)
      {
      if (all_input[i].type === "text")
      /* If this is a text box */
         {
         if (all_input[i].value === "")
         /* If this input box has not been completed */
         /* "if (all_input[i].value.length == 0)" is an alternative
i.e. when the length of the input value is 0 */
               {
               alert("You have not filled in a text box.");
               completed = false;
               /* Set the value of "completed" to false */
               }
         }
      }
      return completed;
      /* Return the value of "completed" (true or false) to the
function validateJS */
}

function validateRadios() {
var years = document.getElementsByName("years");
/* Get all the elements with the name "years" (all the radio buttons)
and put them in a collection also called years */
var radioChecked = false;
/* The value of "radioChecked" is set to false before the for loop
begins and will only be set to true if one of the radio buttons is
checked  */
for (i = 0; i < years.length; i++)
```

```
      {
      if (years[i].checked)
      /* If this radio button has been checked */
         {
         radioChecked = true;
         /* Set the value of "radioChecked" to true */
         }
      }
if (!radioChecked)
/* If not radioChecked - this is the same as saying "if (radioChecked
== false)" */
   {
   alert("You did not select one of the radio buttons");
   }
return radioChecked;
/* Return the value of "radioChecked" (true or false) to the function
validateJS */
}
```

Figure 5: JavaScriptValidation.js

Validation in HTML5 without JavaScript

There are new functions in HTML5 that provide for the automatic validation of certain fields when an attempt is made to submit the form, e.g. when the user clicks the Submit button. The browser support for these new facilities is a little patchy. Different browsers may fail to support different features in the example code we have provided. Modern versions of Google Chrome and Opera may support this functionality.

The validation features in the following example relate to the range of numerical values that are acceptable, e-mail address validation and the checking of URL formats (HTML5 assumes they will start with http:// or https://). There are two other examples of HTML5's new features that can be incorporated into forms, which are not strictly validation. One involves the choice of a date and the other the choice of a colour. Such choices from menus can, however, help to reduce errors.

Here is the HTML5 code with no JavaScript and no significant use of CSS in this particular case.

```
<html>
  <head>
      <title> HTML5 Validation </title>
      <meta charset="utf-8" />
      <link href="style1.css" type="text/css" rel="stylesheet" />
  </head>
  <body>
      <h1> HTML5 Validation Controls </h1>
            <p> Please answer the following questions so that we can
provide a better service for you. </p>
            <form>
            <fieldset>
            <legend>Tell us about your degree: </legend><br />
            <label for="studyTime">What <u>p</u>ercentage of your
```

```
time do you spend on your studies ? </label><br />
            <input type = "number" name = "studyTime" id =
"studyTime" min = "5" max = "100" step = "5" value = "20" accesskey =
"p" autofocus /><br /><br />
            <label for="degreeRating"><u>R</u>ate your degree
programme on a scale of 1 to 5</label><br />
            1<input type = "range" name = "degreeRating" id =
"degreeRating" min = "1" max = "5" step = "1" accesskey = "r" />5<br
/><br />
            <label for="email">Type your e-<u>m</u>ail
address.</label><br />
            <input type = "email" name = "mail" id = "mail" accesskey
= "m" required /><br /><br />
            <label for="webaddress">Type your <u>W</u>eb address
(starting with http://).</label><br />
            <input type = "url" name = "webaddress" id = "webaddress"
accesskey = "w" /><br /><br />
            <label for="date">When did you <u>s</u>tart your degree
?</label><br />
            <input type = "date" name = "date" id = "date" accesskey
= "s" required /><br /><br />
            <label for="colour">Select your favourite background
<u>c</u>olour. </label><br />
            <input type = "color" name = "colour" id = "colour"
accesskey = "c" /><br /><br />
            </fieldset>
            <p>
            <input type = "submit" name = "submit" value = "Submit"
/>
            <input type = "reset" value = "Clear" />
            </p>
            </form>
  </body>
</html>
```

Figure 6: HTML5Validation.html

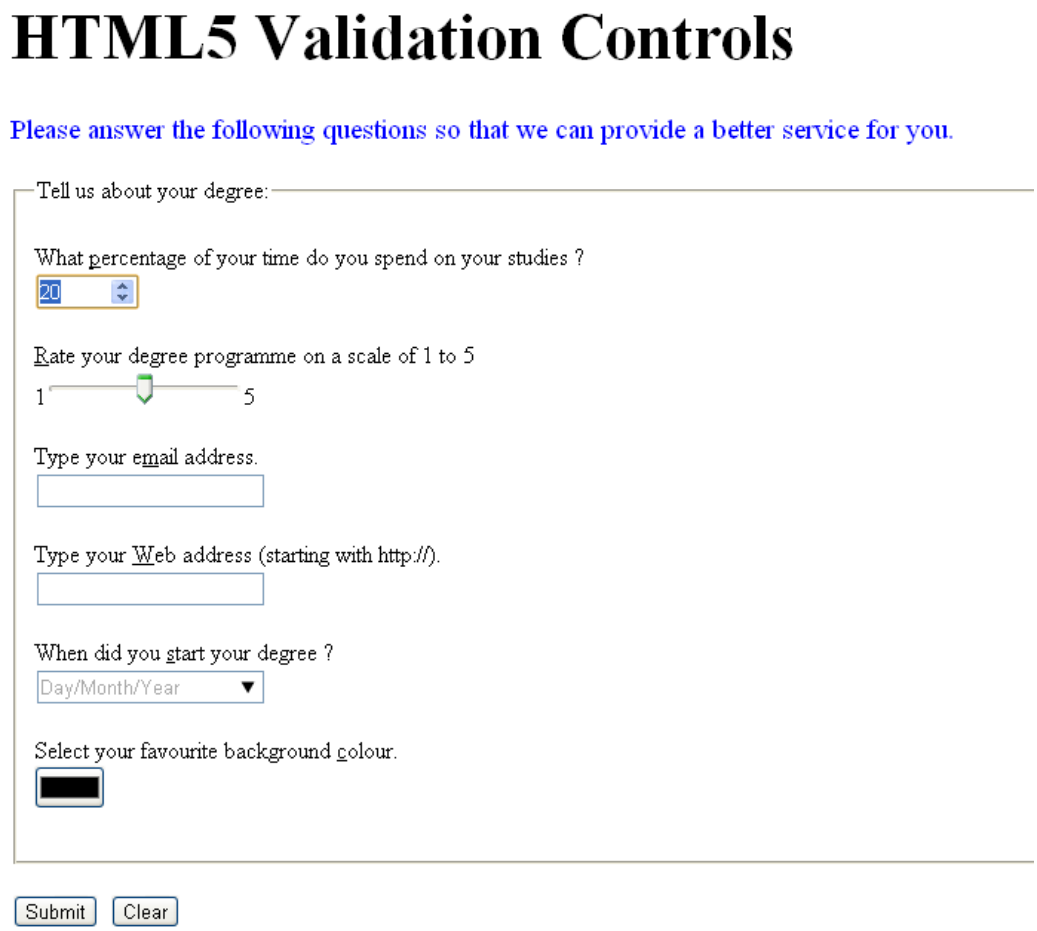

Figure 7: Output from HTML5Validation.html

There are a number of points worth highlighting.

The term *accesskey* has been used in a number of fields in the example (Figure 6). This enhances the usability of the form for users who are not able to easily use a mouse. The *accesskey* provides a way of shifting the focus onto a particular field using the keyboard. The programmer defines the letter that will be used for each field. In IE, Chrome and Safari, the ALT key should be held down while the specified key is tapped. Other browsers use different conventions. You are warned that some ALT-key combinations clash with functions provided by the browser itself, and in these cases the browser will overrule the programmer. You should thoroughly test all accessibility functionality..

The <u> (underline) tag is permitted in HTML5 although it should be avoided as much as possible to maintain the separation of content and presentation. An exception has been made in this case so that the *accesskey* can be highlighted in the field's label. The underlining provides a visual reminder of which key should be used to switch the focus onto a particular field.

The *autofocus* is set in the first field of the form, meaning that the focus will automatically be placed on that field when the Web page loads and will be ready to receive input. The use of *autofocus* must only be used once per form.

In the first field, the new *number* input type has been used, and minimum and maximum values for input have been specified. The expression *step = "5"* means that the menu will present the numbers 5, 10, 15, 20, etc.; *value = "20"* specifies the initial value that will be shown in the menu. Clicking on the arrows will allow this figure to be increased or decreased by the *step* value.

As in all these cases, the way these functions are implemented is dictated by the particular browser we are using, although they are likely to look similar where implemented. If a browser does not support any of these new functions, they are likely to be represented as a text box.

The second field uses the *range* input type, and this is coded in a similar way to *number* although the slider varies visually between browsers. The arrow keys should work with both these new input types if the mouse cannot be used.

The *email* input type means that when the form is submitted, any input will be checked to see if it is in a valid format for an e-mail address. This may not be 100% foolproof, but it is a useful initial check. The error message that is displayed when the input is in the wrong format is browser dependent. In this example, the field has been marked as *required*. Any *required* fields that are left blank when the user submits the form will generate an error message.

The *url* input type will check to see if the input is in a valid format for a Web address. HTML5 assumes that addresses will begin with http:// or https://, so 'google.com' will not be enough to pass this validation test.

A calendar will pop up where the *date* input type is supported. Similarly, the *color* input type will pop up a colour palette from which the user can choose.

From the programmer's point of view, the advantage of this built-in HTML5 validation is that it cannot be turned off or blocked by personal firewalls, which can be a problem with JavaScript. When it is widely supported, it is likely to be very useful. In the meantime, the owners of Web sites have the usual dilemma of deciding whether to risk the negative effects of users visiting a site where functionality is not supported by their own browser. In situations such as an intranet, where the owners can dictate the kind of browser that will be used, there will be fewer problems.

Manipulating strings

JavaScript provides a number of built-in functions for manipulating strings of characters. A string might be thought of as an array of characters where the first character in a string variable called *firstname* can be referred to as *firstname[0]*. This means that a *for* loop can be used to search through all of

the characters in a string. As with an array, the length of the string (the number of characters) will be stored in the variable *firstname.length*.

There are functions to convert all characters in a string to uppercase or lowercase. This is useful if we are comparing two strings but we are not concerned about whether the input contains capital letters or not. We would say that this comparison is not case sensitive. A user might type 'Web' or 'web', but we are not bothered which. To make sure we accept either case, we can convert the string we are looking for (*requiredString*) and the string that is input by the user (*inputString*) like this:

```
if (requiredString.toUpperCase() === inputString.toUpperCase())
   {
   alert ("You typed the correct word.");
   }
```

In this example, both *requiredStrin*g and *inputString* have been temporarily converted to capital letters so that we can make a comparison that is not case sensitive. If we want to store the string that is all uppercase, we should have to say something like:

```
requiredStringCaps = requiredString.toUpperCase();
```

A practical application would be where we were checking a person's username, but we were not concerned about whether the letters input were uppercase or lowercase. The function *toLowerCase()* works in a similar way.

In the following example we are going to extract a reference number from text input which could consist of a sentence or paragraph. Suppose that we know that our reference numbers will all begin with the letters ZYX followed by 6 numerical digits. We can begin by setting up a little HTML form with a textarea in which users can type anything they like as long as they include their reference number.

```
<!DOCTYPE html>
<html>
<head>
<title> Searching a String </title>
<meta charset="utf-8" />
<script type = "text/javascript" src="extractRefNo.js"></script>
</head>
<body>
<form>
<fieldset>
<legend> Tell us about yourself and include your Reference Number.
</legend>
<textarea name = "refInput" id = "refInput" rows = "6" cols = "60">
</textarea>
</fieldset>
<fieldset>
<legend> This your Reference Number. </legend>
<textarea name = "refNo" id = "refNo" rows = "1" cols = "15">
</textarea>
```

```
</fieldset>
<p>
<input type = "button" id = "getRefNo" value = "Click to Extract the
Reference Number" onclick = "getRef()" />
</p>
</form>
</body>
</html>
```

Figure 8: extractRefNo.html

Here is the JavaScript file that contains the function *getRef()*:

```
function getRef() {
var refNo ="";
var refPosition = 0;
var refInput = document.getElementById("refInput").value;
/* Get the input from the first textarea */
refPosition = refInput.indexOf("ZYX");
/* Get the index position of the start of the string "ZYX" (the
position in the long string that forms the total input) */
if (refPosition != -1)
/* If the string "ZYX" is not found, the function refInput.indexOf
will return the value -1 */
   {
   refNo = refInput.slice(refPosition,refPosition + 9);
   /* We are going to slice or extract from the total input the
reference number consisting of "ZYX" plus 6 numerical digits. The
first parameter is the index position of the start of our reference
number. The second parameter is the start position + the total number
of characters we want to extract, which will be 9 in total */
   document.getElementById("refNo").value = refNo;
   /* We display the extracted reference number in the second HTML
textarea called "refNo". The contents become equal to the value
stored in the JavaScript variable we have also called refNo */
   }
else
   {
   alert("The input does not contain a reference number beginning
with ZYX.");
   /* This happens when the function refInput.indexOf returns -1 */
   }
}
```

Figure 9: extractRefNo.js

The comments in the JavaScript code above should explain what is happening. Finally, here is a screenshot of the HTML page showing that the reference number has been successfully extracted from some sample input and displayed in the second textarea.

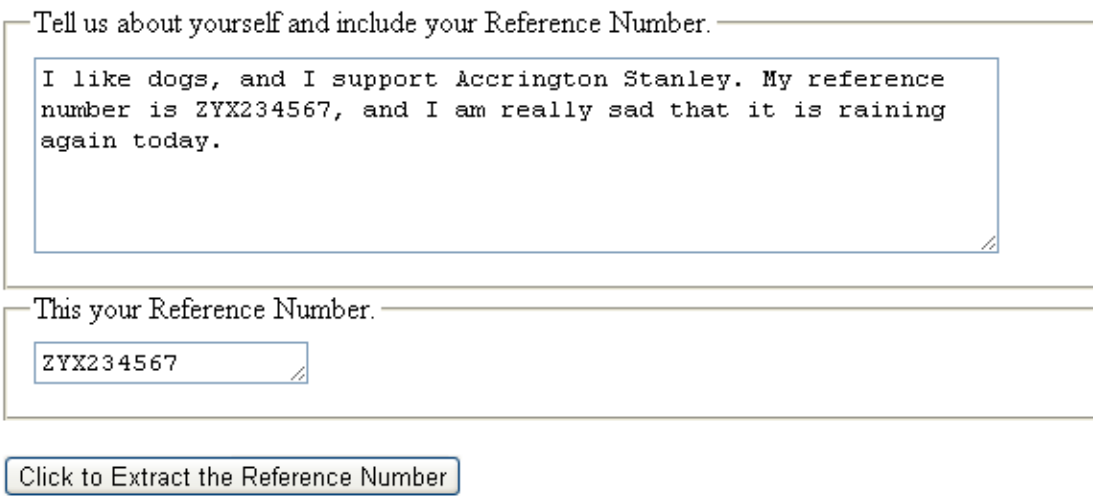

Figure 10: Output from extractRefNo.html and extractRefNo.js

JavaScript libraries

Much of what we want to do with JavaScript is fairly common to all organisations. It is therefore not surprising that sophisticated libraries of JavaScript functions have been developed. Examples are jQuery (2012), Dojo (2012), Yahoo User Interface Library (2012) and Prototype (2012).

Ideally, this code should work with all modern browsers and should be free, open source software. Most of the JavaScript libraries aspire to this ideal. They typically involve the user either downloading or linking to a .js file that contains all of the functions in the library. With jQuery and some other libraries this is not just a case of copying and pasting code into our JavaScript programs or just calling up a function from the library. jQuery has its own shorthand code for making use of its libraries, and learning that code does involve a certain amount of time.

Using such libraries involves a commitment to learning a new kind of notation that is different from pure JavaScript. On the other hand, many people find jQuery and Dojo extremely convenient to use and feel that they save a great deal of time. Using tried and tested code in the libraries may lead to fewer bugs creeping into JavaScript programs. If we are making a decision on behalf of an organisation to commit completely to using such libraries, a great deal of thought will have to be given to making the business case.

We shall be taking a closer look at a little jQuery in Week 7.

For the purposes of our module, you should write as much of your own code as possible in order to demonstrate that certain skills have been acquired. When a small amount of JavaScript code is reused, you should acknowledge the source (to avoid difficulties with plagiarism) and provide sufficient detailed explanation of how it works (usually in comments) to convince your Instructor that you are not copying code you do not understand.

References

Dojo (2012) *Documentation* [Online]. Available from http://dojotoolkit.org/documentation/ (Accessed: 30 August 2012).

jQuery (2012) *Tutorials: how jQuery works* [Online]. Available from http://docs.jquery.com/How_jQuery_Works (Accessed: 30 August 2012).

Prototype (2012) *Prototype JavaScript library* [Online]. Available from http://prototypejs.org/ (Accessed: 30 August 2012).

W3 Schools (2012) *JavaScript RegExp object* [Online]. Available from http://www.w3schools.com/jsref/jsref_obj_regexp.asp (Accessed: 30 August 2012).

Yahoo User Interface Library (2012) *Why YUI ?* [Online]. Available from http://yuilibrary.com/ (Accessed: 30 August 2012).