Programming the Internet—Lecture Notes

Week 3: Introduction to JavaScript and the Document Object Model

Starting to program

This module does not assume any previous programming experience, so we shall start from the beginning. A program consists of a set of instructions that we want a computer to carry out. The language that computers really understand is *binary code*, but it would be very tedious for us humans if we had to write all our programs in binary.

To help us out, some very clever people came up with what we call high-level programming languages that enable us to write our instructions in English-like statements (and it is true that the main programming languages in the world are written in English). Behind the scenes, these instructions are translated into binary code by sophisticated programs known as *compilers* and *interpreters* so that computers will understand what we want them to do.

A compiler translates programs into binary before the program is executed. This gives us compiled code. An interpreter does the translation line by line as the program is running. This is interpreted code. JavaScript is an interpreted language. We do not have to worry about compiling it. The translation is done automatically for us by a browser's JavaScript engine (a piece of software). We often refer to this as *client-side processing* because it takes place within the client (the browser) rather than on a Web server (*server-side processing*). A program is said to run or execute when it starts to carry out the instructions we have specified.

Note: These engines have often differed from browser to browser, giving slightly different results, but are now becoming more standardised. Detailed guides for the JavaScript language can be found in the Mozilla Developer Network (2012a, 2012b). The ECMAScript standards are at the Ecma Web site (2011).

When running a program, we shall usually want to store different pieces of information in temporary areas of memory. At one time, programmers actually needed to know something about memory locations, but it is now much simpler than that. We provide a name or identifier for a piece of information, and where it is stored in memory is decided for us.

Let's take an example. Suppose we want to add together two numbers. It would be more useful to write a program that would add any two numbers rather than, say, just 2 + 7. In programming, we can do this by declaring two variables. Let us call them *numberOne* and *numberTwo*. In JavaScript we would write the following:

```
var numberOne, numberTwo;
```

They are called *variables* because they can contain a value that can 'vary.' Variables can be thought of as containers for storing information. In this case, the number that is stored in a variable such as *numberOne* can be (almost) anything we want. It is good programming practice to declare variables in this way so the program can set aside memory to hold the information we want to store temporarily. JavaScript, unlike some other languages, does not insist on this specific declaration of variables, but it makes the program easier to understand if you adhere to this naming convention.

Many programming languages insist that we specify the type of information that is stored in variables, but JavaScript does not. It is said to be a 'loosely typed' language. Behind the scenes, JavaScript does store information using a number of different data types by making intelligent guesses about the kind of data we are using. The main data types are *number* (integers or real/floating point numbers), *string* (a collection of characters) and *Boolean* (something that is either true or false). There are specific ways in which the JavaScript programmer can insist that data be stored as a certain type, but for the most part we shall not need to worry about this in our introduction to the language.

JavaScript variable names are known as *identifiers* and must begin with either a letter or the underscore character. They may contain only letters, numbers, the underscore and the dollar sign, so most punctuation marks, including the hyphen, are not allowed. Identifiers are case sensitive, so *Pay* will be taken to be a different variable from *pay*. We should also not use any reserved word, something that is part of the JavaScript language (JavaScripter, 2008).

We now have two variable names declared, *numberOne* and *numberTwo*. Next, we want to give them each a value prior to our adding two numbers together. To achieve this, we need an assignment statement. We are said to assign a value to a variable. Let's do two together.

```
numberOne = 4;
numberTwo = 6;
```

These statements will store the number (value) 4 in the variable *numberOne* and the number (value) 6 in *numberTwo*. If read out loud, we should say, '*numberOne* becomes equal to 4; *numberTwo* becomes equal to 6'. In JavaScript, a single equals sign does not mean *is equal to*. To visualise what is going on, imagine that we wrote the following:

```
numberOne ← 4;
numberTwo ← 6;
```

The value on the right is being stored in the variable on the left. Unfortunately, the ← symbol was not available in the character sets around in the early days of programming, so we are stuck with the = sign to denote assignment.

For 'equals' in mathematics, we should write ==

To check whether strings are the same, it is safer to use === (strict equality, which does not see the *number* 20 and the *string* “20” as being equal).

To complete our small example, let us add our two numbers together and save the total in a variable called *totalnumber*.

```
var numberOne, numberTwo, totalnumber;
numberOne = 4;
numberTwo = 6;
totalnumber = numberOne + numberTwo;
```

The last line should be read as either ‘*totalnumber* becomes equal to *numberOne* plus *numberTwo*’ or ‘Add the number (value) stored in the variable *numberOne* to the number stored in the variable *numberTwo* and save the result in (assign the value to) the variable *totalnumber*.’. The calculation on the right of the = sign is made first, and only then is a value stored in the variable on the left of the = sign. The number that will be stored there will be 10 (4 + 6).

External JavaScript files

Now that we have been introduced to the basics of programming, we are ready to look at how we introduce JavaScript into the HTML page. One approach is to use `<script>` and `</script>` tags to contain JavaScript code in the Web page itself. Just like CSS, JavaScript can be embedded in a Web page or can be contained in an external file. We shall try to adopt the good habit of separating HTML and JavaScript so that, to a great extent, one file can be edited or updated without needing to amend the other. Therefore from the start we shall place our JavaScript code in an external file with the extension .js.

Many of the most practical examples of the use of JavaScript involve HTML forms, with which we became familiar last week. In order to understand our first full JavaScript example, we need to look at how we can capture the input from HTML forms.

The Document Object Model

Each part of an HTML file (every pair of tags) is an object that we can manipulate using JavaScript. We can read what the user has typed in a text box or what selections he or she has made from a menu or set of radio buttons.

We have seen that an HTML file contains a series of tags within tags. The `<html>` and `</html>` tags (also known as elements) might be thought of as at the top of a hierarchy or family tree. Within these tags (and below them in the hierarchy) will be tags such as `<head>` and `<body>`. Below the body tag will be `<h1>` and `<p>`. This is the *Document Object Model* (DOM). By giving these elements an *id*, we can conveniently identify and manipulate them in useful ways.

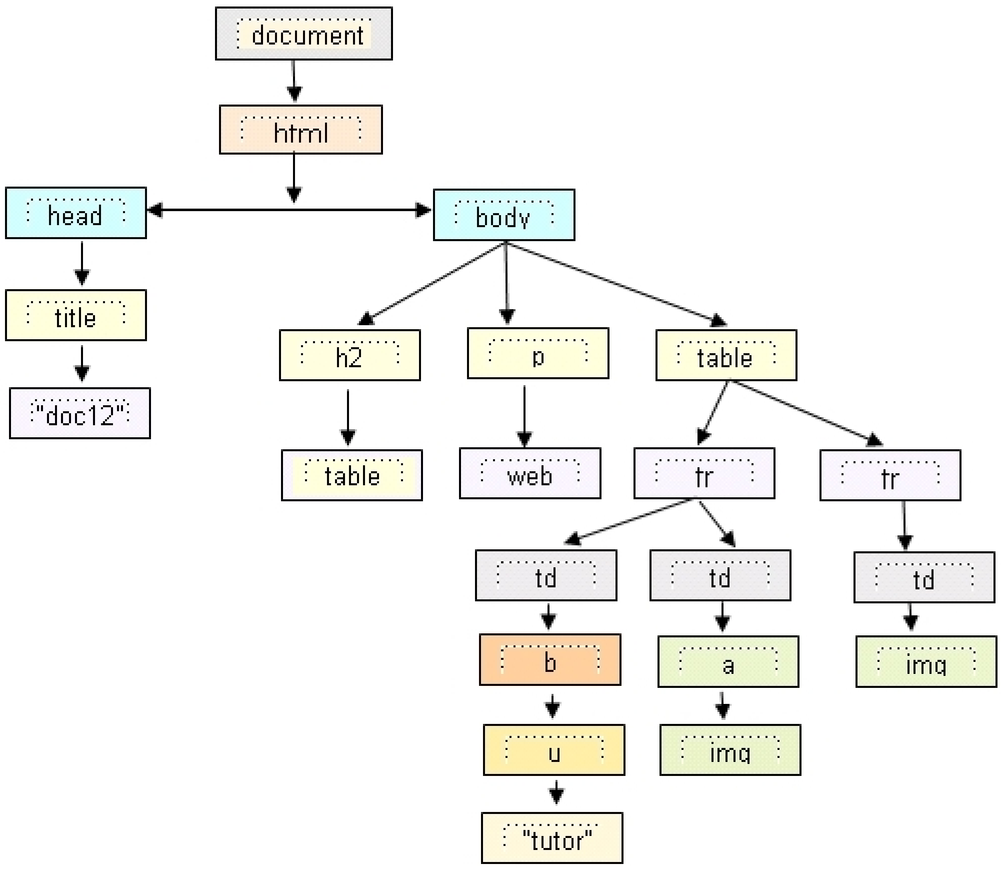


Figure 1: The Document Object Model (DOM)

At one time, each browser had its own DOM, and that was obviously a challenge to programmers trying to get scripts to work in different browsers. Fortunately, the World Wide Web Consortium (W3C, 2009) has produced a standard version of the DOM, which is well supported by modern browsers. This is another example of how a standards-based approach can benefit the Web. An introductory tutorial on the DOM can be found at W3 Schools (2012).

Getting information from an HTML text box

Let us now take a look at how JavaScript can read and manipulate the contents of an HTML text box. In the Week 2 Lecture Notes we defined the following text box within a form:

```
<input name = "surname" id = "surname" type = "text" size = "20"
maxlength = "30" /><br />
```

To identify this text box in the JavaScript, we are going to use its *id*, which is *surname*. A list of a few things we can do in JavaScript follows, with explanations included as JavaScript comments. Note that JavaScript accepts comments in the same form as CSS files. Remember that HTML comments use different opening and closing tags.

```
document.getElementById("surname").value = "";
/* Get the value of the element that has the id "surname" (grab the
contents of the text box) and assign to it an empty string (we make
sure the text box is blank or empty). Another way of saying this is
that the value of the text box called "surname" becomes equal to ""
(a string with no characters). */

document.getElementById("surname").value = "Smith";
/* Put the name "Smith" in the text box with the id "surname" by
assigning a string of characters to the variable value. */

document.getElementById("surname").focus();
/* Put the cursor at the beginning of the text box so that the next
thing the user types will be in that box. */

var thisName = document.getElementById("surname").value;
/* Declare a variable called thisName and store the current contents
of the text box "surname" in that variable. */

var thisNumber =
parseFloat(document.getElementById("average").value);
/* The contents of a text box will be in the form of a string of
characters. Sometimes we shall want this content to be treated as a
number to use in subsequent calculations. In this example, a text box
with the id "average" is having its contents converted to a floating
point number (one with a decimal point) by the function parseFloat
before it is assigned to the variable thisNumber. There is also a
parseInt function to turn text into an integer (whole number). */
```

The *document* object we refer to in document.getElementById("surname").value is the HTML file. *getElementById* is a method, that is, something we can do with the *document* object. JavaScript is an object-based language rather than a full object-oriented language like Java, but getting used to talking about objects and methods will do no harm in preparing for any potential future studies in programming. At this stage just remember that JavaScript and Java are distinct languages although there are some similarities in their syntax.

Event-driven programming and DOM scripting

Event-driven programming occurs when the actions performed by the program are in response to events. Typical events would be whatever the user does with the mouse or keyboard. These events are different from looking at the data the user has input. Programming here consists of event detection (establishing that something has happened) and event handling (deciding what to do in response to a particular event). The code that is written to respond to an event is referred to as an *event handler*. DOM scripting is a form of event-driven programming.

JavaScript provides quite sophisticated methods of detecting events. When an event handler executes, it has full access to the DOM and, therefore, to all elements on a Web page. The properties of elements can be quickly changed by JavaScript code.

Event-driven attributes include *onclick* (when something is clicked with the mouse), *onblur* (when something loses the input focus, such as when the cursor is moved from a text box) and *onload* (when a document has been opened in a Web browser). Details of more event attributes are contained in Chapter 13 of the course text.

Attributes such as *onclick* and others are associated with HTML tags. Here is an example of how *onclick* works with two text boxes. Firstly, the HTML:

```
<!DOCTYPE html>
<html>
<head>
<title> Adding Two Numbers </title>
<meta charset="utf-8" />
<script type = "text/javascript" src="addNumbers.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
= "addNumbers()" />
</p>
</form>
</body>
</html>
```

Figure 2: addNumbers.html

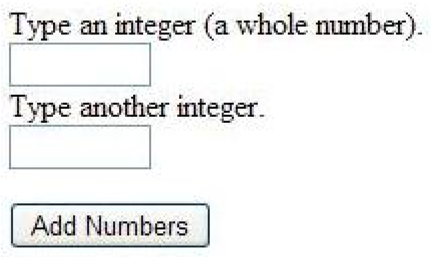


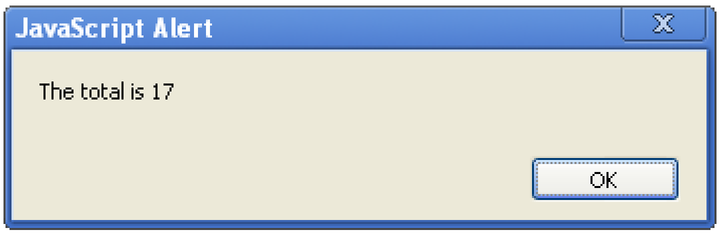
Figure 3: Output from addNumbers.html

Here we have an HTML form which invites the user to type an integer in each of the two text boxes provided. There is a button which, when clicked, will add the two integers together. The text in bold in Figure 2, **`onclick = "addNumbers()"`**, directs the browser to execute some code in a JavaScript function called *addNumbers()*. Earlier, within the `<head>` tags we have pointed the browser to an external JavaScript file called *addNumbers.js*, and this is where it will expect to find the function. Here are the contents of that file:

```
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);
}
```

Figure 4: addNumbers.js

There are a few things worth explaining here. Below we shall look at what a function is, but here in Figure 4 we are making use of *getElementById* to capture the contents of the text boxes and store them in variables called *numberOne* and *numberTwo*. These variables are added together, and the sum is stored in a variable called *total*. The function *alert()*, which is part of the standard JavaScript language, places a box or window on screen with the required message contained within the brackets in the code after the word *alert*. Entering 5 and 12 as the two integers and clicking the Add Numbers button will produce the following alert box:



```
The total is 17
```

Figure 5: Alert box produced by addNumbers.js

The code `alert ("The total is " + total);` needs an explanation. In this example, we can see that bits of information can be joined together. A literal string and a variable are involved here. The words “The total is ”, contained within quotes will be reproduced exactly as they are. Where a variable name is used, the value stored in that variable will be printed. The value stored in the variable *total* is 17; therefore, the output displayed in the alert box will be “The total is 17”. Do not forget to put spaces inside the quotation marks (in this case after the word ‘is’) to separate the literal text from the value in the variable that will be displayed.

Addition and concatenation

The plus sign (+) in the previous example is used for joining strings of characters together to form our output on screen. This is known as *concatenation*.

It may be a little confusing that the same symbol is used for mathematical addition and concatenation. JavaScript decides what to do by looking at the content of the items on either side of the plus sign. If both are numbers or numbers are stored in the named variables, then it will be interpreted as a mathematical addition. Thus, *numberOne* + *numberTwo* is interpreted as being addition as both variables contain numbers, but *numberOne* + *“ and ”* is concatenation (the joining of strings of characters) as the item before the plus sign is a number but the item after the plus sign is a piece of text. Note that in the expression *2* + “2” the item before the plus sign is a number, but the item after the plus sign is a string. Numbers in quotes will be treated as strings. Thus, in this example, JavaScript will concatenate and the result will be a string “22”.

Other mathematical symbols that are commonly used in JavaScript are the following:

```
* multiplication
/ division
- subtraction
== equal to
!= not equal to
=== strictly equal to
!== not strictly equal to
> greater than
>= greater than or equal to
< less than
<= less than or equal to
++ increment
(incrementing will add one to an integer – a whole number; it may,
for instance, be keeping a count of how many times a program has done
something)
```

JavaScript, and any other client-side scripting in browsers, can be disabled by the user. Strict settings in personal firewalls such as those provided by ZoneAlarm or Norton may also disable scripts. It is therefore helpful to use HTML’s `<noscript>` tag to display some message if the JavaScript is not allowed to execute.

```
<script type = "text/javascript" src="addNumbers.js"></script>
<noscript>This is a script to add two numbers. If it will not work,
please check to see if JavaScript is disabled in your browser.
</noscript>
```

Introducing functions

Sometimes, we shall want to execute certain sections of our JavaScript code several times, and in that case it will be convenient to place that code inside a function. By using the function's name within our JavaScript program, we can execute all the code within that function. In our example the function is called *addNumbers()*, and it is contained in the external JavaScript file, *addNumbers.js* . Here is what we call the 'function declaration':

```
function addNumbers () {
var numberOne, numberTwo, totalnumber;
numberOne = 4; numberTwo = 6;
totalnumber = numberOne + numberTwo;
alert("Adding " + numberOne + " and " + numberTwo + " together gives
a total of " + totalnumber + " .");
}
```

If we want to 'call' this function (use its code), we should just write *addNumbers()* in our JavaScript program.

Note that the code within the function declaration is enclosed in braces (curly brackets). We should be familiar with these from our study of CSS. Function declarations, not surprisingly, start with the word 'function'. All functions must be declared, usually within an external .js file before they can be used. Do not forget to tell the browser which external JavaScript file it should look for by including a line of code within the <head> tags of the HTML file. In this case:

```
<script type = "text/javascript" src="addNumbers.js"></script>
```

More objects to manipulate

We have looked at ways in which we could identify and manipulate information in a text box of an HTML form. Let's now take a brief look at the radio button, checkbox, and drop-down menu. With the radio button, the HTML code might be as follows:

```
<input type = "radio" name = "SiteQuality" id = "SQ1" value = "5" />
<label for="SQ1"> Strongly Agree </label> <br />
```

We can test in JavaScript whether this radio button has been selected (checked):

```
var SQ1radio = document.getElementById("SQ1");
if (SQ1radio.checked)
{
// Define some action here
}
```

We shall explain more fully how an *if* statement works in the next section. Note that single line comments in JavaScript can use //, but multi-line comments need /* and */. With a checkbox, part of the HTML code might be as follows:

```
<input type = "checkbox" name = "ItemsOwned" id = "IO1" value =
"laptop" /> <label for="IO1"> A laptop computer </label> <br />
```

We test whether this checkbox has been selected in a similar way:

```
var IO1checkbox = document.getElementById(″IO1″);
if (IO1checkbox.checked)
{
// Define some action here
}
```

This is the HTML for a drop-down menu that allows just one selection:

```
<select name = "country" id = "country" size = "1">
<option value = "NotAnswered" selected = "selected"> Not Answered
</option>
<option value = "Canada"> Canada </option> <option value = "Spain">
Spain </option>
<option value = "Nigeria"> Nigeria </option> <option value =
"Britain"> Britain </option>
<option value = "Australia"> Australia </option>
</select>
```

The JavaScript to detect whether an option has been selected would be slightly different.

```
var country = document.getElementById(″country″).value;
if (country === ″Canada″)
{
// Define some action here
}
```

If the programmer wants to check or uncheck a radio button using JavaScript, he or she would write the following:

```
var SQ1radio = document.getElementById(″SQ1″);
SQ1radio.checked = true; /* To check the radio button (checked
becomes equal to true) OR */
SQ1radio.checked = false; // To uncheck the radio button
```

The checkbox works in the same way.

If statements

Sometimes in our programs, we need to do something under one set of circumstances but something else under a different set of circumstances. We may need to test whether someone is under 21 years of age, male or female, or willing to register details on our Web site or not. What action our program carries out may depend on the answer. It is an important way in which we respond to our customers' needs and provide content that is relevant to their circumstances and interests.

In programming, we often use an *if* statement to test whether one condition, or a set of conditions, is met before we decide what to do next. We may want to say that *if* one set of conditions is true, do *x*, or *else* do *y*. In JavaScript we might say:

```
if (age >= 18)
{
alert ("You may vote");
}
else {
alert ("You are not old enough to vote");
}
```

If the person's age is greater than or equal to 18, the person is old enough to vote (in the UK, US and many other countries). In all other cases (*else*), the person is not old enough to vote. Note that the actions we want to be performed in both the *if* and the *else* part of this statement are enclosed in braces (curly brackets).

Sometimes, we may want to test to see if a series of conditions are met. In this example, we suggest a URL (Web address) where people can find out more information about their browser:

```
if (browser === "Internet Explorer")
{
alert ("Visit http://www.microsoft.com/windows/internet-
explorer/default.aspx");
}
else if (browser === "Firefox")
{
alert ("Visit http://www.mozilla-europe.org/en/firefox/");
}
else if (browser === "Google Chrome")
{
alert ("Visit http://www.google.co.uk/chrome");
}
else if (browser === "Opera")
{
alert ("Visit http://www.opera.com/");
}
else if (browser === "Safari")
{
alert ("Visit http://www.apple.com/safari/";
}
else {alert ("Sorry, we have no information on other browsers.");
}
```

As soon as the program finds a condition that is true, it will execute the appropriate *alert* code and not look at any of the other parts of the *if* statement. The way the braces (curly brackets) are laid out may differ in other JavaScript code examples you see. Here, they are lined up so that we can check that opening and closing braces are present in each part of the *if* statement, but this is a matter of personal preference.

More events

Earlier we were introduced to *onclick* as a way of detecting when the *click* event has taken place, e.g. when a button has been clicked. Sometimes a user (possibly someone with a disability) will be navigating between elements on the Web page by using the *tab* key. As this key is tapped, each element will in turn be highlighted in some way to show that it has the focus. When the *enter* key is tapped while a button has focus, the *click* event will fire, so *onclick* does not only work with a mouse.

There are many events that JavaScript can detect, and the language provides a number of event attributes besides *onclick* to alert us to the fact that an event has taken place. The following example makes use of *onload*, *onsubmit*, *onchange*, *onmouseover*, *onmouseout*, *onfocus*, *onblur* and *onkeyup*. There are plenty of comments in the code, so it should not need further explanation.

Perhaps the best way of understanding what this example code does is to run the scripts by opening the HTML file. Remember that all the code for these examples is contained in a zip file within the online classroom.

First of all, here is the HTML code that calls the JavaScript attributes for detecting events:

```
<!DOCTYPE html>
<html>
        <head>
            <title> Demonstrating JavaScript Event Programming
</title>
            <meta charset="utf-8" />
            <script type = "text/javascript"
src="eventExamples.js"></script>
        </head>
        <body onload = "onloadWelcome()">
        <!-- The load event is fired when the page is loaded/reloaded
or opened -->
        <form onsubmit = "onsubmitMessage()">
        <!-- The submit event is fired when the Submit button is
clicked or when the Submit button has focus and the <Enter> key is
pressed -->
        <fieldset>
            <legend> What is your favourite colour ? </legend>
            <select name = "favcolour" id = "favcolour" size = "4"
onchange = "onchangeselectMessage()">
            <!-- The change event is fired when the contents of an
<input>, <select> or <textarea> element in an HTML form are changed.
-->
            <option value = "Red" selected = "selected"> Red
</option>
            <option value = "Blue"> Blue </option>
            <option value = "Green"> Green </option>
            <option value = "Yellow"> Yellow </option>
            </select>
        </fieldset>
        <p>
        <input type = "button" id = "dangerButton" value = "Do Not
Click!" onmouseover = "dangerMessage()" onfocus = "dangerMessage()"
onmouseout = "normalMessage()" onblur = "normalMessage()" />
        <!-- The mouseover event fires when the mouse cursor is over
the button and the mouseout event fires when the cursor is moved off
the button. The focus and blur events provide similar facilities for
users employing the tab key to navigate between the HTML elements
rather than the mouse. -->
        </p>
        <p>
        <label for="firstName">Type your first name.</label><br />
        <input name = "firstName" id = "firstName" type = "text" size
= "10" maxlength = "20" onblur = "onblurMessage()" /><br />
        <!-- The blur event is fired when the input field loses focus
e.g. when the cursor is moved elsewhere. -->
        <label for="secondName">Type your second name.</label><br />
        <input name = "secondName" id = "secondName" type = "text"
size = "10" maxlength = "20" onkeyup="capturekeyTap(event)" /><br />
        <!-- The keyup event is fired when a key is tapped - strictly
speaking when the key comes up after being pressed down. -->
        </p>
        <p>
        <input type = "submit" name = "submit" value = "Submit" />
        <input type = "reset" value = "Clear" />
        </p>
        </form>
        </body>
</html>
```

Figure 6: eventExamples.html

And now the JavaScript file, which is mentioned within the `<head>` tags of the HTML file. It contains the functions we have written that will be executed when an event is detected.

```
function onloadWelcome() {
alert ("Welcome to this file demonstrating examples of JavaScript
event programming.");
}

function onchangeselectMessage() {
var favouritecolour = document.getElementById("favcolour").value;
if (favouritecolour !== "Red") /* When checking to see if two strings
are equal, it is more reliable to use === (strict equality) or !==
for checking if they are not equal. */
   {
   alert ("Thank you for not choosing red.");
   }
}

function onblurMessage() {
var firstname = document.getElementById("firstName").value;
if (firstname !== "") /* If the firstName field is not blank (empty)
*/
   {
   alert ("It is about time you got around to filling in this
field.");
```

```
   }
}

function onsubmitMessage() {
var favcolour = document.getElementById("favcolour").value;
var firstname = document.getElementById("firstName").value;
var secondname = document.getElementById("secondName").value;
alert ("Your favourite colour is " + favcolour + ". Your name is " +
firstname + " " + secondname + ".");
}

function capturekeyTap(evt) {
var keyTapped = String.fromCharCode(evt.which);
/* The variable "which" is part of the JavaScript language and it is
used to capture the numeric value of the keyboard key that has been
pressed (usually a unicode number). The JavaScript function
String.fromCharCode converts this numeric value into the letter that
was pressed.
This is reliable for letters of the alphabet and numbers not typed
using the numeric keypad. Browser support might be patchy. Try Chrome
or Safari. This function uses a parameter called evt (more about
parameters in Week 4).
The event that has been captured (in this case the key that was
tapped) has been passed to this function. */
alert ("You just typed the letter " + keyTapped);
}

function dangerMessage() {
document.getElementById("dangerButton").style.background ="#ff0000";
document.getElementById("dangerButton").value = "Danger ! Warning !";
}

function normalMessage() {
document.getElementById("dangerButton").style.background ="";
document.getElementById("dangerButton").value = "Do Not Click !";
}
```

Figure 7: eventExamples.js

CSS alternatives to JavaScript

In the example that follows, we show how CSS can be used to create buttons that can be clicked and to detect a limited number of events. In this case, no JavaScript is used at all. This can be an advantage in cases where you suspect users might block or turn off JavaScript in their browsers. These CSS techniques can be used alongside JavaScript, so it is possible to get the best of both worlds to some extent. The code is adapted from an example in McGrath (2009).

Firstly the HTML. You will see shortly that we are attempting to implement a menu of buttons arranged horizontally along the top of the page using a set of links within an unordered list.

```
<!DOCTYPE html>
<html>
```

```
        <head>
                  <title> CSS Events and Buttons </title>
                  <meta charset="utf-8" />
                  <link href="style10.css" type="text/css"
rel="stylesheet" />
        </head>
        <body>
        <ul id = "buttonMenu">
        <li class = "cssButton"><a href =
"https://www.google.com/intl/en_uk/chrome/browser/" tabindex = "0">
Chrome </a></li>
        <!-- tabindex = "0" means that any reluctant browsers will be
encouraged to include these links in the list of items that will be
highlighted by the tab key (often used to navigate by people with
disabilities). As the value is 0, no tab order or priority is
specified here. -->
        <li class = "cssButton"><a href =
"http://windows.microsoft.com/en-gb/internet-explorer/downloads/ie"
tabindex = "0"> IE </a></li>
<li class = "cssButton"><a href = "http://download.cnet.com/Apple-
Safari/3000-2356_4-10697481.html" tabindex = "0"> Safari </a></li>
        <!-- In the Safari browser you will need to change the
default setting to get tab key navigation to work properly. Go to
Preferences, then Advanced, and select the checkbox labelled "Press
Tab to highlight each item on a webpage".  -->
        <li class = "cssButton"><a href = "http://www.mozilla.org/en-
US/firefox/new/" tabindex = "0"> Firefox </a></li>
        <li class = "cssButton"><a href =
"http://www.opera.com/download/" tabindex = "0"> Opera </a></li>
        </ul>
        <h1>The headline has to be positioned precisely below the
menu or it will be partly obscured by it.</h1>
        <form>
        <p>
        <label for="firstName">Type your first name.</label><br />
        <input name = "firstName" id = "firstName" type = "text" size
= "10" maxlength = "20" /><br />
        <label for="secondName">Type your second name.</label><br />
        <input name = "secondName" id = "secondName" type = "text"
size = "10" maxlength = "20" /><br />
        </p>
        <p>
        <input type = "submit" name = "submit" value = "Submit" />
        <input type = "reset" value = "Clear" />
        </p>
        </form>
        </body>
</html>
```

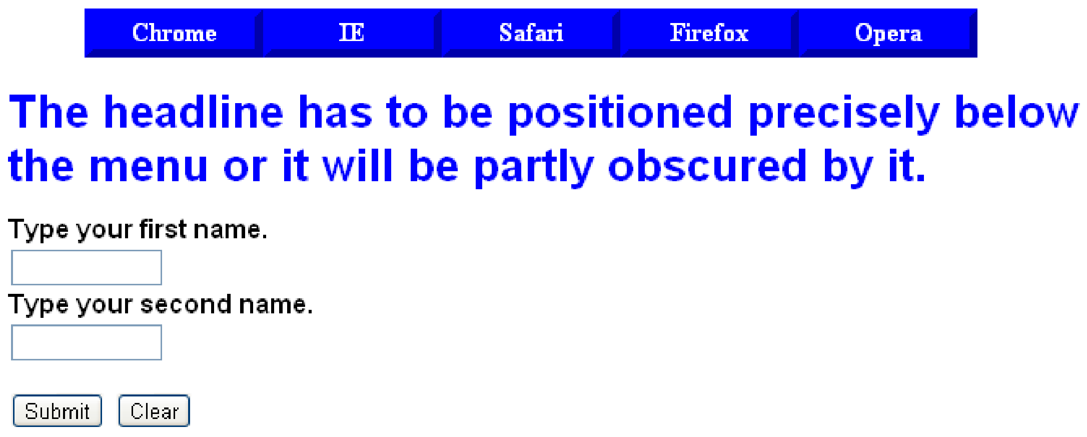
Figure 8: cssEvents.html

In this case, most of the work is being done by the CSS file, which is mentioned within the `<head>` tags of the HTML file. Here is a closer look at the CSS file.

```
#buttonMenu {
position: absolute;
top: 0px; /* The buttons will be at the very top of the page, 0
pixels from the top */
left: 15px;
margin: 0;
}

li.cssButton {
display: block;
float: left;  /* This ensures that the menu items are displayed side
by side. */
margin-right 5px;
text-align: center;
list-style:none;
}

li.cssButton a {
display: block; /* This ensures, in effect, that the entire element
which contains the link is clickable and not just the text that is
associated with the link. */
color: white;
font-weight: bold;
width: 100px;
text-decoration: none; /* The link is not underlined. */
background: blue;
border: 5px outset blue; /* "outset" gives the impression that the
button is raised, but the effect is browser dependent */
}

li.cssButton a:hover { /* When the mouse cursor hovers over the link
*/
background: red;
}

li.cssButton a:focus { /* When the link has focus, such as when
someone is using the tab key to navigate the page. */
background: red;
}

li.cssButton a:active { /* When the link is clicked or when it has
focus and the <enter> key is pressed. */
background: cyan;
}

h1 {
position: absolute; /* Absolute positioning means the headline or
other text which follows the button menu has to be positioned
precisely (in this case 30px from the top of the page). Otherwise it
would be displayed partly behind the buttons at the top of the page.
*/
top: 30px;
color: blue;
font-size: 20pt;
font-family: Arial, sans-serif;
}

form {
position: absolute;
top: 110px; /* The form will be 110 pixels from the top of the page
*/
}
```

```
label {
font-weight: bold;
color: black;
font-size: 12pt;
font-family: Arial, sans-serif;
}

#firstName:focus, #secondName:focus { /* When one of the text boxes
receives the focus, the background will change to silver. */
background: silver;
}
```

Figure 9: style10.css



The headline has to be positioned precisely below the menu or it will be partly obscured by it.

Type your first name.

Type your second name.

Figure 10: Output from cssEvents.html and style10.css

Opening the HTML example file and experimenting with it will provide some working examples of what CSS alone can achieve in terms of event-driven programming. The comments in the above CSS file should also help to explain what is going on.

It is worth noting that, in this example, absolute positioning is used for each element to locate it at a precise number of pixels from the top of the page *(position: absolute;)*. The advantage of absolute positioning is that elements can be placed exactly where the programmer wants them to be. The disadvantage is that on different devices with very different screen sizes, this inflexible approach can place elements outside the viewable area of the device, resulting in excessive scrolling.

Debugging JavaScript code

In learning HTML, we made use of the W3C Validator to check for mistakes in our code. There is no equivalent W3C tool for JavaScript. However, the JavaScript Lint tool should be of use (SourceForge, 2012). The easiest way to use it is to copy and paste JavaScript code onto its Web page (The Online Lint) and click the Lint button.

It is possible to get most browsers to display some error messages that provide good clues as to what has gone wrong. Errors are one of three types:

1. Syntax errors: The programmer has not followed the rules of the language. It is with this type of error that the Web browser's messages will be of most help.

2. Runtime errors: The programmer has followed the rules (produced correct syntax) but has done something that causes the script to crash (such as dividing by zero). Hopefully, these introductory programs will not have this type of error.

3. Logic errors: The script runs but produces the wrong result. The logic is faulty. It may be something as simple as typing the wrong variable name when attempting to make a calculation or finding that a totally unexpected value is being stored in a variable at a crucial point in the program. There is limited automated support for identifying this kind of error, but during debugging it is often useful to use a series of *alert* statements showing the values stored in variables to reassure ourselves that they have the values we thought they had at different parts of the program.

Each browser also provides some support for debugging (Harris, 2009, p.90-117):

- Internet Explorer: When JavaScript code has syntax errors, a small exclamation mark in a yellow triangle may be shown in the bottom left of some versions of this browser. Double click on this symbol, and a window with an error message will appear. This will make reference to a line number where it is believed the error occurs. For some recent versions of IE there are developer tools which can be accessed by pressing the F12 function key.

- Firefox: To see error messages in Firefox, select the browser menu Tools → Error Console (Windows) and click on the link that refers to your script. There is a free JavaScript debugging add-on or plug-in (extra piece of software) for Firefox called *Firebug* (Firefox, 2012). If you use this browser, it should be worthwhile downloading the plug-in.

- Safari: (Used widely on Macs). It has a console window which will display errors if it is open before the JavaScript is run. Ensure that the Develop menu is showing and then select Develop → Show Error Console. You might first need to go to Preferences → Advanced and check the box 'Show Develop menu in menu bar'.

- Google Chrome: A JavaScript console may give some useful clues. It can be accessed by selecting the menu options Tools → JavaScript Console (although older versions of Chrome might differ).

- Opera: It also has an error console that can be opened by selecting Tools → Advanced → Error Console.

When files are not in the same directory

In all the programming examples we have used, it has been assumed that the relevant HTML, CSS and JavaScript files are all in the same directory. This will not always be convenient. We shall no doubt be creating sub-directories and placing many of the files we use in separate directories.

We could use the full Web address (URL) for a directory we are accessing from a browser, but it is usually easier to employ a shorter version known as a 'relative path'. A relative path specifies where a directory is in relation to the current directory. This is also more convenient if we are moving files and directories from one machine to another. We do not want to have to keep changing the addresses of the files to which we are linking. A relative path specifies where the other directory is relative to the current directory. We might be using an HTML file that is, for example, calling some CSS or JavaScript code held in a .css or .js file. Let us look at a small example where we have a home directory on a Web server with three sub-directories.

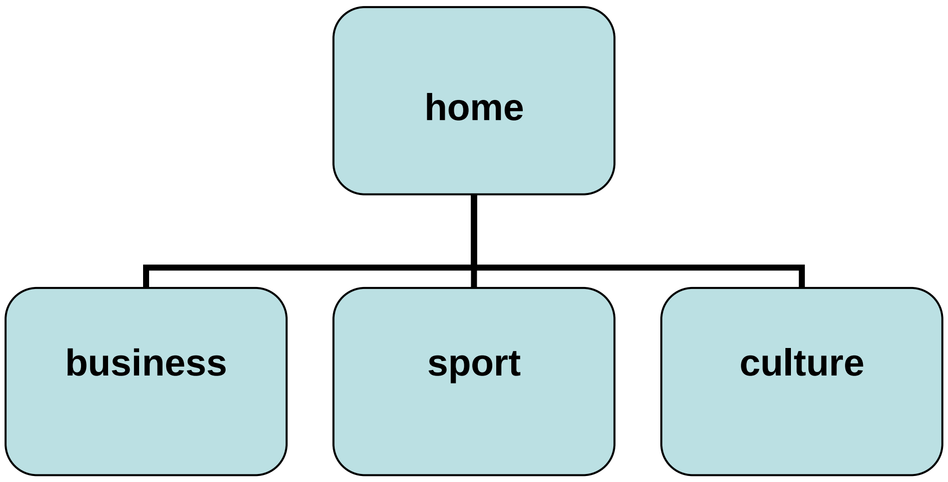


Figure 11: Web Directory Structure

Let us assume that our Web home page, which is called *index.html*, is in the *home* directory. We decide to use an external CSS file called *mystyles.css*, which is stored in its *business* sub-directory. We should then include this line of code in the `<head>` tags of our home page.

```
<link href="business/mystyles.css" type="text/css" rel="stylesheet"
/>
```

Now, let us look at the situation where an HTML file called *contacts.html* in the *business* sub-directory needs to link to a JavaScript file called *events.js* in the *culture* sub-directory. These directories are at the same level. They are both sub-directories of the *home* directory. In navigation terms, we need to first go ‘up’ one level to the *home* directory. This move is denoted by the two dots (..) in the path below. We must then go ‘down’ into the *culture* sub-directory. The line of code that will link to *events.js* in the file *contacts.html* will be as follows:

```
<script type = "text/javascript" src="../culture/events.js"></script>
```

Finally, let us assume that an HTML file, *holidays.html*, is in the *culture* sub-directory, and it wants to link to another HTML file called *cities.html*, which is in the *home* directory. Now, we are going ‘up’ the directory structure, and we have already learned that this involves the use of those two dots that we discussed earlier. A link in the file *holidays.html* might look something like this:

```
<a href="../cities.html"> My favourite cities </a>
```

Remember that these code examples are going to be executed by a Web browser; therefore, the paths to particular files are, in effect, shortened Web addresses that we can use where files are on the same server. Because they are a kind of Web address, we use forward slashes in naming directories rather than the backslashes used by Windows.

References

Ecma (2011) *ECMAScript language specification* [Online]. Available from http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf (Accessed: 31 August 2012).

Firefox (2012) *Firebug* [Online]. Available from https://addons.mozilla.org/en-US/firefox/addon/firebug/ (Accessed: 24 August 2012).

Harris, R. (2009) *Murach’s JavaScript and DOM scripting.* Fresno, California: Mike Murach & Associates.

The JavaScript Source (2012) *About the JavaScript Source* [Online]. Available from http://www.javascriptsource.com/jss-about.html

JavaScripter (2011) *Reserved words in JavaScript* [Online]. Available from http://www.javascripter.net/faq/reserved.htm (Accessed: 24 August 2012).

McGrath, M. (2009) *CSS in easy steps.* Southam, United Kingdom: Easy Steps Limited.

Microsoft (2012) *Discovering Internet Explorer developer tools* [Online]. Available from http://msdn.microsoft.com/en-us/library/gg589507(v=vs.85).aspx (Accessed: 24 August 2012).

Mike Murach & Associates (2012) *JavaScript and DOM scripting: FREE download of chapters 2 and 3* [Online]. Available from http://www.murach.com/books/mdom/chapters.htm (Accessed: 24 August 2012).

Mozilla Developer Network (2012a) *JavaScript guide* [Online]. Available from https://developer.mozilla.org/en-US/docs/JavaScript/Guide (Accessed: 24 August 2012).

Mozilla Developer Network (2012b) *JavaScript reference* [Online]. Available from https://developer.mozilla.org/en-US/docs/JavaScript/Reference (Accessed: 24 August 2012).

SourceForge (2012) *JavaScript lint* [Online]. Available from http://www.javascriptlint.com/index.htm (Accessed: 24 August 2012).

W3C (2005) *Document object model (DOM)* [Online]. Available from http://www.w3.org/DOM/ (Accessed: 24 August 2012).

W3 Schools (2012) *HTML DOM tutorial* [Online]. Available from http://www.w3schools.com/htmldom/dom_intro.asp (Accessed: 24 August 2012).