Unit 4 is worth 10% of your portfolio grade

Podcast - Introduction to Unit 4

In this unit we will be asking you to find some JavaScript code that someone else has written and incorporate it seamlessly and effectively into your site. The code could be anything from a menu system or other navigation aid to an animation or game, but it should be appropriate to and support the scenarios and personas you have already developed in Unit 1.

You will also be asked to read and attempt to understand the code you are using, and to critique and (if possible) improve or modify it in the light of your critique. As a result of this, you will gain several useful competences and insights (we hope), including a better idea of what is possible with JavaScript, a fairly good idea of JavaScript commands and construction, and a better sense of how technologies might enhance a site.

Learning Outcomes

When you have completed this unit, you should be able to

- critique JavaScript code written by others, identifying examples of both good and bad practice.
- use JavaScript to add dynamic content to pages.
- modify existing JavaScript code to extend and alter its functionality and, where appropriate, to correct errors and cases of poor practice.

Problem

Important: do not begin this task until you have completed Unit 0, 1, and 2, including reflective learning diary entries.

Paying extremely close attention to the needs and scenarios and personas of Unit 1, find some small piece or pieces of JavaScript code that might be useful from the Web (e.g., from javascripts.com—see the links for this course for potential sites or find your own, and add them to our collection on the Landing).

Make sure that you have a right to use the code: it should be public domain or licenced as open source that you are free to use and modify. Do ensure that you abide by whatever conditions are required (more on this below).

Try to avoid anything particularly complex—stick to scripts that you are going to be able to understand, at least in outline.

Incorporate it into one or more of your web pages.

Very important: for this and for all occasions on this course when you will be re-using or modifying code written by someone else, make sure it is properly ascribed—there should be no doubt about where it came from in the marker's mind. We strongly encourage legitimate re-use of code, but we will treat plagiarism, in which you do not correctly cite the author, very seriously and punitively.

Information: How to Cite Code

In many cases where the licence is open source, the code itself will include information about the attribution required. Often, this will require you to retain author information and, notably in the case of GPL (General Public License), a link to the full licence conditions and original source site.

Typically this will be in a comment within the code, but there may be included files that you must also provide. Some authors will require that you provide a link on your page that is visible in the web browser and/or other information. Under no circumstances should you ignore or circumvent such conditions. Read them carefully, and ensure that you comply with the author's licence terms. Failure to correctly cite code may be treated as plagiarism.

If there is no immediately obvious licence information required, such as when code is public domain or part of a free tutorial, you should, where known, cite the author's name and, whether or not you know who the author might be, provide the precise URL (not just the site name) where it can be found as well as the date on which you downloaded it. Minimally this

should be provided as a comment each place that the code is used, but where it does not interfere too much with your page content, it is also a good idea to provide a visible citation on the web page itself. In the unlikely event that you choose to use non-online code, follow APA guidelines for citation, information on which may be found at http://www2.athabascau.ca/services/write-site/apa.php (Athabasca University Write Site guide to documentation).

Explain what the code does in as much detail as possible. An example is provided in the **Process Guide** section of this unit.

Provide a critique of that code, indicating how and why it is good and/or bad. Even if there is nothing obviously wrong, you might minimally explain that the code is properly indented, well-commented, making good use of variable names, etc. More information is provided in the **Process Guide** section.

If necessary, using the code that you have already incorporated into your page, modify it so that you increase its functionality, and correct any errors you find in structure, style, or content.

In your reflective learning diary, explain how the code improves the experience of the personas you created in Unit 1, and how it helps with the scenarios you presented. If, as a result of this, you need to make changes to the documentation you created for Unit 1, you should explain what changes were made and why you made them in your learning diary.

Do not modify the original documentation—create a new copy of the modified version with your learning diary for this unit.

As always, what you choose and how you incorporate it should be driven and informed by your personas and scenarios. You will lose marks if there is no clear connection here—to give an extreme example, if your site is intended to support a funeral business, then you would be marked down for selecting code that let people play games (especially *hangman*).

Process Guide

Choosing the Code

As well as ensuring that the code does something that your site users would find valuable, for your own sake it is wise to stick with small, easy-to-understand code examples. A sensible length of program to look for would be between 20 and 30 lines long at most (not counting blank lines or instances of HTML or CSS). The code you use should, however, be of sufficient complexity to enable you to provide a critique and summary. You may choose slightly shorter or longer examples if you wish, bearing in mind that increased length leads to an increased workload for you.

It's a good idea to find some well-commented code: comments can be identified as either single lines that start with two slashes (//) or as larger block of code delimited by slash-star and star-slash (`/* ---some comments -- */`). Well-written code always has comments to help other programmers to understand what it is doing. In fact, more often than not, it is to help the original programmer remember what the code is about.

> **Something to remember for the rest of the course**: Always include plenty of comments in your code unless you have an extremely good reason not to. For this course, there will never be a good reason not to provide comments!

Be careful to avoid anything that is too long and complex: if it contains more than 30 lines (not including comments and blank lines) it is almost certainly too much work to decipher. Also beware of things that require a great deal of integration to use effectively on your site, unless it is something you really, *really* want, in which case be prepared to put in some effort with debugging.

Critiquing and Reading the Code

As always, think about what the users of your site are like, what you want your site to do, and the kind of activities users will want to engage in. Choosing pieces of JavaScript involves a mix of top-down intentions and discovery of the possibilities of the tools, which may inspire ideas to link what you want to do with how it is done.

In order to do this exercise effectively, you are going to have to learn how JavaScript works, at least in terms of broad programming constructs and logic. We do not need nor expect you to know very much just yet! Bear in mind that you will be spending a lot more time on this throughout the rest of the course, so you do not need to know more than the basics right now.

This is not a small task, but it is one that you will be continuing throughout the rest of this course, so do not worry if it seems a bit confusing at this point. We provide some helpful links, but do feel free to explore further (it is an important and primary skill to be able to do this, so the more practice you get here the better) and, as always, add links to the course bookmarks on the Landing if you find something useful.

We strongly advise that you spend some time investigating the basics of JavaScript at various tutorial sites and similar resources. A good start might be the Wikipedia article at http://en.wikipedia.org/wiki/JavaScript. If you prefer to watch and listen, then you might find Douglas Crockford's great series of videos helpful: http://yuiblog.com/crockford/. Another very popular and well-designed resource is W3Schools, which provides an excellent range of tutorials from beginner through to expert—see http://www.w3schools.com/js/default.asp. You may find these of continuing value in the next unit, in which you are required to actually write JavaScript programs.

We don't expect you to understand more than a small part of the code that you will be using here (yet) and will not penalize you for missing bits or getting things a little wrong, but we think the process of looking through it will provide a helpful foundation for the next unit in which you will be creating your own code. With that in mind, here are a few things to look out for and approaches to reading the code that may be helpful.

Here is a very simple example of the kind of thing we are looking for an explanation and critique:

The original code:

TOP

```
function helloworld (userName){

alert("hello"+userName);

return;

}
```

The explanation and critique:

The function takes a parameter of 'userName' and displays an alert box that says "hello <username>", where <username> is the value passed to the function. It would be used to provide a personalized greeting on a site, though the pop-up alert would probably be too annoying to use on a real site.

The code itself is quite well written, but it would be improved by adding a comment to explain what it does. Also, the statements in the statement block are not indented, so it is hard to see where the block begins and ends. The lines should be indented to improve readability. Also, the function name would be better in camelCase, again to make it easier to read.

Of course, it is highly unlikely that you would find this particular function useful in any of your pages! You might get good marks for your explanation and critique, but you would almost certainly lose them again for choosing code that would not fit your scenarios or personas. The examples you discover are likely to be a bit more complex and lot more useful.

The kinds of snippets that might be valuable could include (by way of example—feel free to choose your own):

- image rollovers or slideshows
- menu systems – pull-down, expandable, etc.
- form validation (e.g., to ensure correct entry of phone numbers, dates, etc.)
- browser detection (and a means of doing something as a result such as showing or hiding a section)
- games
- calculators/conversion tools
- theme pickers
- visual embellishments (e.g., animations – but use with care!)
- databases
- sortable lists
- etc.

There are endless opportunities and examples to pick but it is important that, whatever you choose, you can justify it in terms of your site purpose, design, themes, personas and scenarios.

As in the previous two units, there is no formal feedback for this unit given at this point, but you are strongly encouraged to look at and comment on the work of others on the course, if available, and to request feedback from your tutor.

(Almost) Everything You Need to Know about JavaScript for This Exercise

Do not confuse Java and JavaScript. The two have some very superficial similarities in syntax but are completely different languages with different origins, ways of working, and capabilities. The word 'JavaScript' was chosen primarily for marketing reasons because of Java's then popularity and the resemblance, beyond a

few bits of basic syntax shared by many other languages like C and PHP and Perl (often known as “curly bracket languages” because they have—well—curly brackets), mostly ends there.

The standardized version is officially called ECMAScript (which keeps the distinction very clear), but virtually no one uses that term. Microsoft has its own slight variant known as Jscript, which is largely the same thing. All code we will deal with here should run in multiple browsers, so we will strongly discourage anything specific to a certain vendor and will always use the generic term 'JavaScript' for the coding being performed on this course.

Like HTML and CSS, JavaScript is written as plain text and (at least for the purposes of this course) it runs in a web browser—you do not need any other software to use it. Like CSS, it can be used within the HTML of the document or, more usually, created in a separate file that is linked to in the HTML document, especially once you start to want to reuse code or to make longer programs. You will probably recognize the similarity with CSS here: the command to include a script is very similar in form to that used to link to a CSS file:

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

There is nothing to stop you including JavaScript directly in your HTML, surrounded by <script> </script> tags although for larger programs, that soon becomes difficult to maintain. Within an HTML document, most of the code you will be looking at will be enclosed in the following tags:

```
<script type="text/javascript">
// some Javascript commands...
</script>
```

TOP

Information: those two slashes (//) signify that what comes next is a comment—something we have written to make it easier for a human to understand the code we have written. JavaScript ignores anything after the slashes when it finds them. If we want to make longer comments, we enclose them in a /* and */ combination. For instance:

```
/*
This is a long comment.
It appears over two lines
*/
```

It is really important to use plenty of comments in your code to explain what it does, why it does it, when it was modified, and so on: this makes it **much** easier for someone else (or

you, when you have forgotten why you did something) to modify and maintain the code. In fact, when writing programs, it is a pretty good idea to start by writing them first in plain English (or whatever language you use) as comments, and only once you have explained what you are going to do—then and only then—add the actual JavaScript.

Note: older JavaScript programs that you come across may instead use the form <script language="Javascript">. This is frowned upon quite seriously nowadays. You should take pains to avoid it and, if using such scripts, replace that attribute with the proper form (type="text/javascript"). Most browsers will still deal with the old attribute quite happily, but it is a far less flexible way to specify what kind of code is to follow and is deprecated in all modern HTML standards. Newer or cut-down mobile browsers may not work if you use this form.

It is also possible to create typically simple JavaScript statements as attributes of tags, using a predefined set of browser events to identify things that happen either automatically as part of the document flow (e.g., <body onload="some javascript command">) or in response to user input (e.g., <a onclick="a bit of JavaScript">). Events are very important to us because they are the things that allow interactivity on the page.

The Document Object Model

For the most part, JavaScript is used to manipulate elements on the HTML page (and/or CSS styles). To do this, it employs the Document Object Model (usually referred to as the DOM) which is a way of describing the document as a hierarchical tree with branches, sub-branches, and leaves. When you have been writing HTML and CSS code you have, perhaps unwittingly, been making use of this DOM quite extensively: every tag and attribute you create is represented internally by the browser as part of the DOM, relative to other parts. The branches and leaves are generally fairly easy to identify because HTML tags have an opening and a closing part that can contain other HTML: so, if something is enclosed within something else, it is further towards the leaf end of the hierarchy than the thing that encloses it. This also helps to explain why it is really important to nest your tags properly when creating well-formed HTML: <h1>this makes no sense </h1> if you are thinking of things containing other things.

There are some special parts of the tree that are not quite as obvious—**document** and **window** are particularly important because they allow us to describe things relative to these main containers. Every browser has a document object model describing what it contains, that JavaScript can read or, mostly, write to. Older versions of the DOM did not allow so much writing, so it was harder to make changes to things on the fly.

Information: Most browsers provide means to let you view the DOM for any given page, either as a built-in feature or an add-on: see http://en.wikipedia.org/wiki/DOM_Inspector for information on getting to this for your browsers. If your browser doesn't have this feature built in, we highly recommend that you add it, so that you can easily identify the elements and attributes of your documents and visualize where they sit in the hierarchy.

Unfortunately, although the W3C has defined a standard that all browsers adhere to to some extent, not every browser uses precisely the same DOM, so JavaScript written for one browser may not always work in another. This is especially true of legacy browsers but even true of newer browsers that may or may not implement particular standards. The growth of HTML 5 may help to reduce that in some ways, but in others, may make things worse because it allows different standards to be used as and when the creators of browsers see fit. Differences in implementation are just something we have to live with: it's annoying, it's inconsistent, and often requires us to write the same routine in more than one version so that we can cater to different browsers, but that's just the way it is!

A Client-Side Scripting Language

JavaScript is a scripting language, which means that unlike a full-fledged programming language like Java, C++, or BASIC, it cannot be used to do everything that is possible for the computer to do: a scripting language is concerned with controlling another program or programs, in this case, the web browser. This accounts for most of its limitations because it can only be used to make the browser do things that the browser is capable of doing and respond to events that the browser can listen for (assuming it runs in a browser—if a script interpreter is available to them, then it can be used by other separate programs such as Flash or even web servers, but we are only interested in web browsers here).

At the time of writing you couldn't, for instance, write a JavaScript program running in your web browser that sniffed packets on a network, changed the settings on your sound card, or wrote to a Microsoft Access database, because web browsers generally don't know how to do those things—yet. However, this is a constantly developing field, and there are proposals to allow some of those things on the table. Some things will never be possible, however, because of the need to ensure security: you would not want a malevolent or inept script to do things on your behalf like access and send bank account details, for example!

Remember: a scripting language is designed to let you create scripts that manipulate the host environment's facilities, in this case the things displayed in a browser.

JavaScript as used here is a *client-side language*: it runs in the user's web browser, not on the web server. In other words, the *processing* of the code happens on a user's local machine, not on the machine that is serving the files. If you want to do processing at the server end, such as database handling, programmed content generation, or capturing user input in a persistent database, you need a different set of server-side technologies (such as those that are covered in COMP 466). This imposes some severe limitations on what you can do.

Essentially, you are only able to modify the experience of a single user, not a whole site, and you are dependent on what browser and settings that user has available at the time. To make things even more constraining, JavaScript is also prevented from accessing some things in the web browser for security reasons. For example, you cannot read or manipulate pages or elements of them that come from a different website with your JavaScript code (if you think about it, you can probably see why—what might happen if you were able to read and manipulate the contents of, say, a banking site that someone had loaded in another tab or Window?).

There are implementations of the JavaScript language that can run on servers, too, as well as very similar client-side languages that do not directly run in web browsers—ActionScript used in Adobe Flash is a good example, but so too are plugins for Mozilla programs (Firefox, Thunderbird, etc.) and widgets or gadgets on Apple and Microsoft desktop operating systems. These versions of JavaScript are beyond the scope of this course, but lessons learned here will be directly applicable to such instances.

JavaScript has traditionally been implemented as an *interpreted* language, which means that there is a program (the JavaScript interpreter) that interprets your code when it runs, turning it into something the machine understands and can run in real time. Some implementations of JavaScript now include a *compiler*, which converts your code into a more efficient, machine-readable form before it runs.

Compiled code can run much faster than interpreted code (the computer has to do less work, and the compiler can take the whole program into account, not just the current lines) but compilation itself is time-consuming. Such systems are therefore usually much slower to start the first time they are run, but run much faster the second time you use them. There are several in-between variants that compile parts of the program for speed but interpret others, or that compile to a faster format but not all the way to the machine code that computers use underneath it all. Others do simpler things like removing comments and spaces so that the computer has to perform a little less work in loading and running the code. You don't need to know much about this yet, and it is not covered any further in this course.

So, for our purposes, JavaScript is a programming language that can be used to manipulate web pages and, to an extent, web browsers, using aspects of the documents exposed via the DOM. Much of what JavaScript does therefore depends upon understanding the DOM (more on that later), but before we can do anything with it, the language itself has to be capable of doing the basics of what all programming languages can do—it has things like commands, variables, arrays, objects, and so on that control the flow of the program and make it perform the manipulations of the DOM that we require.

The Importance of Getting Things in the Right Order

All JavaScript code is executed a statement at a time, in order, unless we use special commands to make it do otherwise—we can break the sequence to make a program loop (iteration), or to do one thing or another based on criteria we supply, or branch to a different place (selection). We'll talk some more about that below.

A statement is usually defined by the fact that it ends with a semi-colon, much like a CSS declaration, though it is acceptable to simply use a carriage return to signify the end of a line (which means you need special tricks to make a JavaScript statement spread over more than one line—in fact, you use the backslash '\' symbol to do that).

And, like a CSS declaration, you can group statements together to form *blocks* of statements using curly brackets. Also like a CSS declaration, JavaScript lets you assign values to variables, though in a somewhat different syntax and for somewhat different purposes. After that, CSS and JavaScript differ considerably. Here is a very simple JavaScript program that writes a line of HTML to the page:

```
<script type="text/javascript">
        document.writeln("<h1>Hello world</h1>");
        document.writeln("<p>This line has been generated by JavaScript</p>");
</script>
```

Here is a slightly more complex program that performs this operation 10 times using a *for loop* (more on that below). Try to work out what is happening:

```
<script type="text/javascript">
     for (i=1;i=10; i++){
          document.writeln("<h1>Hello world</h1>");
          document.writeln("<p>This line has been generated by
          JavaScript</p>");
     }
</script>
```

> **Hint**: whenever you see chunks of code like this, it might be interesting to copy them into an HTML document and open that document to see what happens. You could also try adjusting

some of the code—for instance, changing values or contents of strings. The worst that can happen is that you will get bugs and it won't work.

This is good: a lot of the process of learning to code effectively involves being unafraid to make mistakes and finding out what happens when you make them. Remember, the most essential thing about learning to code is to practice, practice, and practice again. If it worked the first time every time, you would learn a fraction of what you will learn from it **not working**, frustrating though that may be!

Variables

A variable has a value, a bit like HTML attributes or CSS selectors. Sometimes we declare variables before we begin in order to make the code easier to read and easier to change. For example: currentUser="Jon" or maximumValue=6. Notice, incidentally, that the two examples are different: a string variable is enclosed in quotation marks, while a numeric variable is not. This helps the browser to know what it can and cannot do with a variable—it can multiply numbers, for example, and convert a string (the technical term for a sequence of characters such as a word or sentence or telephone number) to uppercase, operations that would make no sense when performed on a variable of the wrong type.

Most of the time, variables are (as the name suggests) variable. In other words, we change the value of them as we run through the program. We might, for instance, use an iteration to increase the value of a number variable each time the loop continues to (say) count the number of key presses a user makes. Or we might base what the program does next on the value of a variable that stores information about which web browser (technically, *user-agent*) the person is using. Or we might reverse the order of words in a string variable sentence or strip out non-numeric characters from a telephone number.

A lot of the time, computer programs are manipulating variables and passing them back and forth between different parts of the code. A lot of the time we cannot know in advance what those values will be (e.g., when users input data or our program is run on an unknown machine), but we do know what kind of values to expect and, logically, what we want to do with them.

For instance, if we are writing a program to convert between Celsius and Fahrenheit, we know that there is a 'temperature' value that we need to manipulate. In fact, we probably know that we will want values for degrees Celsius and degrees Fahrenheit. This is where variables come in. Variables let us give labels to things that we can fill with values. You might think of them as little boxes intended to contain some information that are labelled so that we can find them more easily. For our temperature converter, we would probably create a

variable with a name like 'celsius' and another with a name like 'fahrenheit' that we could use to contain the values entered and what they have been converted into.

Variables never have spaces in their names (though, of course, the values they contain may well have them) and are usually created according to a consistent pattern. Many people like to use camelCase, with separate words indicated by capital letters to make the meaning clear: e.g. myVariable or theResult. Note that JavaScript is a case-sensitive language: 'myvariable' and 'myVariable' are two different variable names. This is often a cause of some nasty bugs!

A bit more about variables.

Arrays of Variables

Sometimes we want to store or manipulate whole lists of values that share something in common—days of the week, months of the year, sets of dogs' names, collections of temperatures over the course of the day and so on. Rather than create a new variable for each similar thing, it is almost always easier and more maintainable to create an **array**—a set of related variables. The details of how these work are for you to discover in the next unit but, for now, it is worthwhile to be aware that, when you see something like 'temperature[1]' with a square bracket containing a variable or number, it is part of a larger set of variables in an array.

Operators

If we are manipulating variables, then we have to have the means to manipulate them. A simple example is the plus symbol. We might use this with variables to say, for instance,

```
theTotal=subTotal1+subTotal2;
```

This assigns to theTotal the value of subTotal1 added to subTotal2. We could use the original value of the variable itself, too, if we like:

```
theTotal=theTotal + 25;
```

Other common operators are * (multiplication), - (subtraction), / (division) and ^ (raised to the power of). You can look up others in a JavaScript reference manual.

You might also come across unary operators that do one very simple task: ++ or -- being the most common. As you might expect, ++ increments a variable by one, while -- decrements it by one. So, if we had a variable called 'myValue' with a value of 5 and wrote a statement myValue++, it would increase the value of myValue to 6. This is useful a surprising number of times, especially in loops, where we might often wish to increase or decrease the value of a counter variable with each iteration of the loop.

Running the sample scripts: if the example scripts below do not work or fail to work when you try to run them a second time, try refreshing the page. This is due to interactions between the scripts and Moodle's own JavaScript implementation.

Sequence, Selection and Iteration

The loop controlling the program flow in the previous example is an instance of *iteration*—repeating something more than once. Typically, we divide the ways of controlling program flow into *sequence, selection*, and *iteration* (though some might also mention branching as a separate category).

Sequence

Sequences are the default way that programs behave. Unless something stops them or redirects them along the way, statements are executed (run, played back) in sequence, in the order in which they are written. The browser reads the statement, runs it (if it can), then moves on to the next one. However (and this is where programs become most useful) they may also loop or branch. In the trade, we tend to talk of these three things as sequence (one thing follows the next), selection (we choose between two or more things to do next), and iteration (we repeat the same thing over and over until some condition is reached where we stop the process).

Iteration

Iterations allow us to run the same statement or group of statements more than once, nearly always varying one or more parameters each time it runs or testing to see whether something has changed. A *parameter* is a variable we pass to the code. So, for example, we might create a loop where, each time it runs, it receives a variable to store a counter that decreases each time the loop runs, until the loop reaches zero, after which the execution will continue with the next line of code after the loop.

Alternatively, we might create a loop that continues to run until, say, the user presses a key to tell it to stop. Loops are powerful and commonplace constructs in all programing languages. When you see words like 'while', 'until', or 'for' as JavaScript commands, you are looking at a loop, and for the most part, you can assume that some iteration is involved. There are some more complex forms such as *recursion*, whereby a chunk of programming code repeatedly calls itself with constantly shifting parameters, but you don’t need to worry much about that yet.

Here are examples of iterations where we have provided you with the means to change a couple of variables to change the behaviour of the loop (note that the actual code is slightly more complex than the simple version that you see here—we have adapted it so that it modifies this document rather than writing over the top of it. If you wish, you can see the real code we have used by downloading the scripts.js file referred to in the <head> section of this document).

An example of iteration: the 'for' loop...

TOP

```
<script type="text/javascript">

/* This script simply displays 'hello world' in increasing font sizes, starting at
the first value of i and ending at the second value, increasing in single-step
increments. You can change the value of the starting and ending points to make
this loop behave differently. Try it! */

for (i=    ; i<=    ; i++) {

/* Notice how, in the next statement, because the quotation mark is a meaningful
character in JavaScript that could be misinterpreted, we prefix it with a
backslash when we want to use it as part of a string, rather than to mark the
beginning or end of it. That way, the Javascript interpreter or compiler knows it
is not part of a Javascript command. This is called 'escaping' - the backslash is
an escape character (the reason is to do with its origins in wireless telegraphy)
that modifies how the next character is treated. */

document.writeln ("<p style=\"font-size:" + i +"px\"> hello world </p>";
   }
</script>
```

An example of iteration: the 'while' loop...

```
<script type="text/javascript">

/* This script reads what you type in a pop-up prompt field, changing the colour
of a piece of text to match what you have typed, until you type 'stop'. Note that
this uses a built-in method that converts a string to lower case and that != means
'not equal to'. It is modifying the contents of a <span> that has an id of "myText
"—in this case, we have applied it to this code sample. The code would work better
if we created it as a do-while loop, and the use of the prompt command is ugly,
but this is just for illustration. */

var myColour="black"; // We need to give it a value to start with
//so the interpreter has something to interpret.
 while (myColour.toLowerCase() !="stop"){
      document.getElementById('myText').style.color=myColour;
```

```
     // The prompt comment is irritating, but it is a quick
     //and dirty way to get input from a user.

          myColour= prompt("Enter a colour or type 'stop'");

 }
</script>
```

Run the script.

Selection

TOP

Selections let the program make decisions based on either other things that are happening in the program or input from the user or other device. For example, we could ask the user a question and, if they say 'yes' execute one chunk of code and, if they say 'no', execute another. Alternatively, we might make the program do something different depending on whether it is running in Internet Explorer or Mozilla Firefox. Or we might perform a calculation and, depending on whether the result is positive or negative, do something different. The range of possibilities of things we can test for and the responses we can make is virtually limitless. When you see words like 'if' 'else' or 'case' in a JavaScript program you are looking at something that involves selection.

An example of selection: the 'if' statement

Here is a simple example of using selection to give a different message depending on what browser a person is using:

```
<script type="text/javascript">

/* This bit of code simply looks at the browser name and, if it is Internet
Explorer, tells you to stop using it, otherwise congratulates you on your
choice. The 'navigator' object is a sign of the origins of JavaScript—it was
originally included as part of Netscape Navigator so that has become embedded
in the language. Notice that we test equivalence using two equals signs (==).
Some languages allow you to use a single equals sign, but JavaScript thinks a
single = means that you are assigning a value to a variable, and returns a true
or false value depending on whether that assignation worked or not. This is not
what we want! */

if (navigator.appName=="Microsoft Internet Explorer"){
alert ("You seem to be using Internet Explorer.
Please get a different browser unless you are just testing this

     page to see whether it works with that browser");
```

```
}else{
     alert ("Good: you appear to be using "+navigator.appName);
}
</script>
```

> **Note**: the message of this program is pretty clear. It is important to support Internet Explorer, so you should use it to test your pages wherever possible, but as a web developer, you would usually be better off using other browsers such as Firefox (especially good because of the many extensions that are available to help you with debugging and developing code), Safari (very good built-in debugging and code development tools—remember to turn them on in your browser preferences), or Chrome (reasonable debugging tools, a fast implementation of JavaScript as well as being very secure, fast, and standards-compliant, so great for testing as you go along). The cardinal rule is, however, to test your code with as many browsers and operating systems as you can. What looks and works great on your machine in a single browser can **often** appear and behave very differently on a different machine or a different browser.

Branching: Functions and Objects

Running the sample scripts: if the example scripts below do not work or fail to work when you try to run them a second time, try refreshing the page. This is due to interactions between the scripts and Moodle's own JavaScript implementation

Although a program usually runs in sequence unless it iterates or we force a selection to occur, it is often convenient for us to interrupt the program flow and jump or branch to another bit of code. We sometimes want to interrupt the sequence and go to another place in the program, especially as a result of a selection or iteration but maybe just because we want to organize our code better by splitting it into logical chunks or because we want to do something similar on more than one occasion. Usually, we want to go back to where we came from afterwards, typically having modified some data or performed some action for the user along the way. Different languages provide different ways of doing that. In JavaScript, this is usually performed using functions.

Functions are separate blocks of quite self-contained code that can be called by programs (including other functions) at any time. Once a function has finished running (usually having changed something along the way

and often returning a value to the calling code) the code from which it was originally called continues on to the next statement after the one that called the function.

For example, we might write a function that converts Fahrenheit to Celsius, then use that every time we need to perform that conversion. Or we might have a very long sequence of statements, and to make it easier to read and maintain, split it into a set of distinct functions that we call one after the other. This makes it much easier to understand the code because we can see how it is structured without having to work all the way through it.

In this sense, you can think of functions as being a bit like paragraphs, headings, and sections in written documents, but they are far more than a simple means of organization and can save a *lot* of time because of how they can be reused in coding. Typically, you will find that JavaScript programs of any length are nearly always constructed out of many functions.

Here is an example of a function being called repeatedly from a loop that shows one way to incorporate JavaScript and HTML. It uses a couple of loops to generate the table

```
<script language="text/javascript">

/* This script takes a number and returns its powers up to 10 (i.e., the result of
multiply the number by itself up to 10 times). It does this by building the HTML a
bit at a time, then writing it all out to the document at the end. Along the way
it calls a function called "myFunction" to discover the power it is looking for.
*/

// Initialize the number—we have made it possible for you to change this to
experiment!
var number=
//Start filling the table variable but don't print anything yet.
var theCode="<table border='1' width='100%'><tr>";

   //Create a row of table cells numbered 1 to 10.
   for (var i=1; i<=10; i++){
//Add the table cells—note the use of \n which inserts a new
//line feed—this makes the resulting HTML easier to read
//if we view the source code of the page

     theCode=theCode+"<td>"+number.toString()+"^"+i.toString()+"</td> \n";

}

  //Close the table row and start a new one.

   theCode=theCode+"</tr><tr>\n";

  //Fill the new row with the powers returned by the function.
   for (var i=1; i<=10; i++){
```

```
//This is all we need to do to call the function, passing 'i' as the parameter.
      //The function returns a string version of the power required.
      theResult=myFunction(number, i);
      theCode=theCode+"<td> "+theResult.toString()+"</td>";
   }
//Close the row and the table.
   theCode=theCode+"</tr></table>";
 //We have created a div called 'functionresult' in our HTML document
       //to take the results.
   document.getElementById("functionresult").innerHTML=theCode;
//Now here's the function:
function myFunction(number, power){
      /* Notice how this function takes a parameter of power, not i - a more
      descriptive label. The function is not part of the loop so it can have
      its own local variables that do not have to match the names of the
      variables it was called with. This makes it much easier to use it in
      many different places and with many different programs, if we want. The
      function returns the value we are looking for (functions don't have to
      return values, but it is useful sometimes) */
      //Create the result using the built-in Math object's power method. var
      theResult=Math.pow(number,power); //Convert the result to a string and return it. return
      theResult.toString();
}
</script>
```

Objects

There is a slightly more complex kind of chunk of code that we often use called an *object* that behaves a little like a function, but it is even better for code maintainability and code re-use. Objects are extremely powerful and elegant tools for constructing complex and reliable programs from small pieces, a bit like a very nerdy form of Lego.

Each object does its own thing and talks to other objects through very well-defined interfaces that tell it what to do or how to behave, or to ask it what it has to say for itself. A really nice thing about objects is that you as a

programmer don't need to know anything about how they work if you are using someone else's objects: you just need to know what they can do, how to make them do things and read what they return. JavaScript uses very many built-in objects and allows you to create your own.

Objects have *properties* and *methods*—these are the parts that they expose to the rest of the code and the means through which we can talk to them. A property is something you can look at and (often) change: for instance, the document object has properties like 'location', 'title', and 'fileCreatedDate' that you can read and/or change. A method is something that an object lets you do to it—a way of telling it to do its stuff. For example a string (text) object has methods like toLowerCase (to convert it all to lower case) or substring (provide just a portion of the string, starting at one point and ending at another).

We highly recommend that you spend time learning about objects, but be aware that the concept requires you to understand a lot of other things before it becomes really useful. If it is too confusing now, wait until the next unit where it should become clearer.

> **Information**: JavaScript has a large number of built-in objects to perform common tasks so that we do not need to write code for them ourselves every time we need them. Most of these map directly to the DOM and are the main means JavaScript uses for allowing us to interact with it.

Wait – There's More

There are many other programming concepts and constructs that you will come across as the course progresses, and this is a tiny overview of some of the more important ones. If you come across others in your investigations, feel free to explore and discover more about them, but don't worry too much if you don't have time for that now.

For now, the small set of examples given here account for a large percentage of what you are likely to find as you start looking at JavaScript programs. As you come across other keywords and constructs, don't worry too much at this stage if a lot of it seems like gobbledegook—it takes a long time to learn a language, and we are only hoping that you begin to get a feel for what JavaScript code looks like, the ways it is used, some of the more common words and grammar that you will find. As you move to the next unit, you will start to see how it all fits together and how you can use it to take control of the way the browser works and behaves.

Don’t Panic!

Do not worry too much about the fact that this will all feel overwhelming for now unless you are a superhuman cyborg genius. The usual process in learning a language, even for those who have learned a lot of them before, is curiosity, bewilderment, confusion, angst, a sense of being overwhelmed, head-banging, suffering, and then flashes of insight, moments of joy as you find a few solutions, and sheer delight when you realize you are in control of the machine, rather than the other way round.

One certainty in all of this is that the most probable way you will gain proficiency is to practice, get it wrong, practice some more, practice some more, keep getting it wrong, and practice some more until, finally, you will find it all makes at least a bit of sense. Programming is a craft, perhaps an art, and like all arts and crafts, takes a lot of practice and perseverance and trial and error to get it right. It’s a bit like learning a musical instrument. We can tell you where to put your fingers, provide a bit of insight into some theory, give you feedback and reassurance and all the other good things a teacher can do, but when it comes down to it, the only way to let those lessons sink in is to try, fail, try, fail, and try again until it works. Practice does, indeed, make perfect.

Indicative Grading Criteria

Grade	Criteria

A	● Critique JavaScript code written by others, identifying examples of both good and bad practice: you should be able to demonstrate an in-depth knowledge of code structure and form, identifying inefficient or unmaintainable code accurately, spotting syntax and logic errors, identifying good and bad practice, considering usability and compatibility issues, and recognizing the distinctive artistry (or lack of it) shown by the programmer. The code you critique should be sufficiently rich to encompass a wide range of programming constructs, functions, classes, and commands, including manipulation and use of page and/or form elements, different methods of sequence, selection, and iteration, use of classes and objects, use of functions, parameter passing, declaration of variables, comments, and different data types. ● Use JavaScript to add dynamic content to pages: Your code should be clear, maintainable, and well structured, suitably integrated into the web page so that it can be easily modified and maintained. ● The code will be carefully chosen and its use thoroughly justified in the context of the personas and scenarios developed in Unit 1. ● Modify existing JavaScript code to extend and alter its functionality and, where appropriate, to correct errors and cases of poor practice: hand-in-hand with the critique, you will have taken examples of rich coding and modified them in a manner that explicitly fits the needs of your site.
B	● Accurately describe what the code you select does. ● Critique JavaScript code written by others, identifying examples of both good and bad practice. ● Use JavaScript to add dynamic content to pages. ● Choose appropriate code that is relevant to the scenarios and personas of Unit 1. ● Modify existing JavaScript code to extend and alter its functionality and, where appropriate, to correct errors and cases of poor practice.

C	● Describe the code in broad terms, mostly accurately. ● Critique JavaScript code written by others, identifying an example or two of good and/or bad practice. ● Use JavaScript to add dynamic content to pages. ● Choose code that does not conflict with the scenarios and personas of Unit 1. ● Small modifications of existing JavaScript code to extend and alter its functionality and minor corrections of errors and cases of poor practice.
D	● Describe what the code does, perhaps rather broadly and with the odd inaccuracy. ● Critique JavaScript code written by others, identifying at least one example of good or bad practice. ● Use JavaScript to add dynamic content to pages. ● Choose code that is broadly justifiable in terms of the scenarios and personas of Unit 1, but adds little of value to the site. ● Limited or negligible modification of existing JavaScript code to extend and alter its functionality.