The Open University

TT284 Web technologies

Block 2 Web Architecture

An Introduction to JavaScript

Prepared for the module team by Andres Baravalle, Doug Briggs, Michelle Hoyle and Neil Simpkins

Introduction	3
Starting with JavaScript “Hello World”	3
Inserting JavaScript into Web Pages	3
Editing tools	4
Browser tools	5
Lexical structure of JavaScript	5
Semicolons	6
Whitespace	6
Case sensitivity	6
Comments	7
Literals	8
Variables and data types	8
Variables as space for data	9
Declaring variables and storing values	10
Expressions	11
Operators	11
Assignment operators	11
Arithmetic operators	12
Comparison operators	12
Logical operators	12
String operators	13
Operator precedence	14
Operators and type conversions	15
Debugging JavaScript	16
Statements	17

Copyright © 2012 The Open University

- Conditional statements 18
- Loop statements 19

Arrays 20

- Working with arrays 21

Objects 22

Functions 23

- Writing functions 24

Variable scope 25

Constructors 26

Methods 27

Coding guidelines 28

- Why do we have coding guidelines? 28
- Indentation style and layout 28
- ‘Optional’ semicolons and ‘var’s 29
- Consistent and mnemonic naming 29
- Make appropriate comments 30
- Reusable code 30

Other sources of information 31

Web resources 31

Introduction

This guide is intended as an aid for anyone unfamiliar with basic JavaScript programming. This guide doesn't include information beyond that required to complete TT284 (there is very little on object orientated coding for example). To use this guide you must be fully familiar with HTML and use of a web browser.

For many TT284 students this guide will be unnecessary or will serve perhaps as a refresher on programming after completing a module such as TU100, perhaps some time ago. It is not intended that you read the entire guide. Instead use the table of contents to identify sections covering aspects of JavaScript you feel unsure about when and if you need them as you work through Block2. Some parts of this guide are duplicated in the Block 2 guides where these are core to an understanding of the practical work.

Within this guide there are some references to the Open JavaScript Guide which is also available from the Module web site or from:

http://sourceforge.net/projects/javascriptguide

As well as this guide there are very many other excellent tutorials, guides, samples and reference texts available on the web. Some of those are listed at the end of this guide and several excellent texts are available online via the OU library (and listed on the TT284 web site).

Starting with JavaScript "Hello World"

There is a long tradition of starting out with any new programming language by implementing a toy 'Hello world' program, so lets' start there. First you need to know how to place JavaScript into an HTML page.

Inserting JavaScript into Web Pages

There are three common ways of inserting JavaScript code in a web page:

1 Inside an HTML tag script, in a CDATA section[1]:

```
<script type="text/javascript">
/* <![CDATA[ */
    alert("JavaScript is working in your browser");
/* ]]> */
</script>
```

2 Inside an external .js file, linked to the HTML page:

```
<script type="text/javascript" src="path/to/file.js"> </script>
```

(The external .js file will include just the JavaScript code, no HTML)

3 As a value of some HTML attributes:

```
<a href="javascript:alert('JavaScript is working in your browser');">link</a>
```

Activity 1 'Hello world' script

You are now ready to try your first Hello World script, which is the typical first program that you learn to do when you study a programming language.

A Hello World program just prints 'Hello world', and nothing else.

Include the following instructions anywhere in the **body** of your HTML page:

[1] A CDATA section, in XHTML, is a portion of a document that should be interpreted as character data (with no markup).

```
<script type="text/javascript">
<!--
    document.write("Hello world");
//-->
</script>
```

JavaScript uses ‘document.write’ to print information into the web page where the script is contained.

After having tried the Hello World script, try to modify it in the following ways:

- try to include the commands in an external JavaScript (ie called ‘some_name.js’) file, as explained on the previous page
- try to modify the third example above to provide an ‘alert’ message using alert instead of document.write.

Editing tools

Whilst you can quite easily write and test JavaScript applications using just any simple text editor and a browser, you may find it useful when writing your code to use some of the features that are included in editing tools for developers.

- **Line-numbering**: this is useful when debugging, to quickly go to specific points of the code.
- **Syntax highlighting**: elements in different categories (for example XHTML and JavaScript) are rendered in different colours. Typically, incorrect code is highlighted in a different colour too, thus making it easier to spot errors.
- **Command completion**: allows the user to type the first letters of a command and see what commands start with that sequence of letters.
- **Code folding**: allows you to hide or show blocks of code.
- **Code formatting**: allows the code to be automatically formatted by the text editor; very useful to manage long files;
- **Context-sensitive documentation**: provides quick documentation while the user is writing code, reducing the number of times required to check reference material.

Tools for Web development are typically divided into WYSIWYG (What You See Is What You Get) editors and text-based editors.

WYSIWYG editors aim to have an interface in which the user can view a web page in a way that is very similar to the final results that will be shown in a browser. As you might have already seen in TT280, WYSIWYG often turns into WYSYHYG (What You See You Hope You Get) because of the different support (and quality of support) in browsers for different subsets of Web standards.

If you are using Microsoft Windows, you might download (http://notepad-plus-plus.org/) and use Notepad++, a simple text-based editor that can help you to write your JavaScript code more efficiently. An alternative to this is the HTML-Kit (**http://www.chami.com/html-kit**), which is also a viable option.

Another editor which has a range of plugin extensions and can be used with various languages is gedit (http://projects.gnome.org/gedit/). Available for Windows and Mac this tool is simple to use but quite powerful.

If you are using a Mac or a Unix-like system, we would recommend you use Bluefish (**http://bluefish.openoffice.nl**) or JEdit (**http://www.jedit.org**, also available for Microsoft Windows).

Web developers often have their favourite tools (and often those tools are not available for free). Apart from text editors focused on Web development, such as HTML-Kit, there are a number of general-purpose text editors (such as Notepad++).

General-purpose text editors can be used to write code in different programming and scripting languages. The main advantage is that you have just one tool, for everything:

XHTML, JavaScript, PHP etc. Eclipse (**http://www.eclipse.org**) is probably the most used general purpose text editor, and is a powerful tool for Web development, but has a quite steep learning curve and I would not recommend it for your first steps into programming.

If you want to explore the most common alternatives, a list of text editors is available from Wikipedia: **http://en.wikipedia.org/wiki/Comparison_of_text_editors**.

Browser tools

Firefox (**http://www.mozilla.com/firefox**) is a Web browser which has a built-in JavaScript console (Tools > Error Console) and a DOM object browser on some platforms. This is particularly useful when used with Chris Pederick's Web Developer extension (**https://addons.mozilla.org/firefox/60**) which allows you to disable JavaScript, cookies, and other useful things quickly and easily.

Later on in this guide, you will be introduced to other tools that can help you to develop and test Web applications.

Activity 2 Try some tools

If you have not already installed a text editor you may like to do so now. If you don't have a favourite of your own then you could use gedit which is available from:

http://projects.gnome.org/gedit

You might well also want to download Firefox and install the Web Developer extension from

https://addons.mozilla.org/en-US/firefox/addon/web-developer

After that, you can try to familiarise yourself with the tools by trying this 'Hello World' script:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
        <title>Hello World</title>
    </head>
    <body>
        <h1>Hello World</h1>
        <script type="text/javascript">
        alert("Hello World");
        </script>
    </body>
</html>
```

If you have some tools you prefer to use please do mention them in the appropriate module forum so that others can try them out.

Lexical structure of JavaScript

Now you should start to learn and to understand the lexical structure of JavaScript (that is, the *meaning* of its terms in common usage).

Learning the lexical structure means understanding the elements that are used when creating scripts with JavaScript.

Semicolons

A JavaScript program is quite simply a sequence of instructions made up of symbols (such as keywords, characters with special meaning, numbers, and text chunks) to which the language associates a special meaning.

Semicolons after JavaScript statements

Simple statements in JavaScript are typically followed by a semicolon (;):

```
var x = 1; var y = 2;
```

You may, however, omit the semicolon if each of the statements is a placed on a separate line:

```
var x = 1
var y = 2
```

You should typically follow every statement by a semicolon and a new line, unless you need to use as little space as possible. Readability of the code is improved by using semicolons after every statement, and it is a requirement in other development languages (e.g. PHP, Java, C).

```
var x = 1;
var y = 2;
```

JavaScript does require either semicolons or line breaks to separate instructions. This means that you could have a JavaScript program with no line breaks at all, just with semicolons. In some cases, this is done to obfuscate the JavaScript code and to make plagiarism harder. (Note: by obfuscating I mean making the code unnecessarily complicated to minimise its readability.)

As JavaScript code used in a website can be easily downloaded (whether the website wants to make it freely available or not), a number of different techniques are used to avoid the code being copied. Don't forget that the fact that you can read the JavaScript code in other websites doesn't necessarily mean that you can use it.

You are, of course, free to read and to study the JavaScript code, but you are allowed to use it in your web pages only if the licensing conditions of the code permit you to do so (in most cases, they will not).

Whitespace

Whitespace can be used to format the code

Whitespace (spaces, tabs and newlines) can be used freely to format and indent the code. You just need to consider that if you insert a new line and the chunk of code that is left in the line appears to be a complete, valid statement, JavaScript will insert an implicit semicolon, possibly altering the meaning of the code.

Consider this code:

```
return
true;
```

which will be interpreted by JavaScript as:

```
return;
true;
```

The code will return the value `undefined` (which is the default value when return is called without a parameter), instead of the value `true`. Don't worry if you don't fully understand this piece of code. For now, I just wanted you to see how inserting a new line can completely change the meaning.

Case sensitivity

JavaScript is case sensitive

Computer languages can be case sensitive or case insensitive (or may be a combination of both). JavaScript is case sensitive, which means that the name of keywords and other language identifiers must always be typed with the correct capitalisation.

Comments

Make an abundant use of comments

Comments are chunks of text which are not executed by the JavaScript but are placed to ensure that the code can be read in an easier way.

You should get into the habit of using comments in abundance, both to make sure that other developers can understand your code (unless you explicitly want to avoid that) and to refresh your own memory when returning to it.

If you are including code from other sources (and if you have the right to do so), you may want to add comments to the code, if they are missing. This will improve the readability for the future.

JavaScript has two ways of inserting comments:

```
// the rest of the line is a comment

/*
This block of text is a comment.
In this way I can comment on multiple lines at the same time.
*/
```

HTML comments have been used for a long time to hide JavaScript from browsers which do not support JavaScript, as illustrated by the following example:

```
1   <script type="text/javascript">
2   <!--
3   document.write("This is a test");
4   //-->
5   </script>
```

Let's examine the code line by line (I have added numbers for ease of reference).

Line 1 starts a JavaScript block.

Line 2 is an HTML comment, making the browsers that don't understand JavaScript ignore the following lines.

Line 3 is a JavaScript statement.

Line 4 is more tricky. The first 2 characters (`//`) are a valid JavaScript comment – so JavaScript will ignore the rest of the line. The first 2 characters, for HTML, will still be part of the comment block started in line 2, which will end with the characters `-->`.

You should use the approach above to insert JavaScript code only when you are using HTML 4 (or earlier).

In XHTML you can use CDATA sections to tell to the browser not to consider the code inside as markup but to treat it as character data. Content included between the `CDATA` opening (`<![CDATA[`) and closing (`]]>`) tags is not parsed by the browser: `&` and `<` inside a CDATA section are not treated as the start of markup.

An example will clarify this. Consider the following XHTML code:

```
<![CDATA[
<author>John Smith</author>
]]>
```

This is equivalent to the XHTML code:

```
&lt;author&gt;John Smith&lt;/author&gt;
```

Returning to our original JavaScript example, I recommend the following approach when inserting JavaScript in XHTML pages (does not apply to HTML):

```
1   <script type="text/javascript">
2   /* <![CDATA[ */
3   document.write("This is a test");
4   /* ]]> */
5   </script>
```

Let's examine the code line by line.

Line 1 starts a JavaScript block.

Line 2 starts a CDATA section, and is included in a JavaScript comment block (which makes the JavaScript interpreter ignore it).

Line 3 starts a JavaScript block.

Line 4 closes the CDATA section, again including the code in a JavaScript comment block.

This is by all means not the only way to include JavaScript in a web page, but it's a quite safe one.

Activity 3 Hiding scripts

You have seen two different approaches to include JavaScript code in a web page: one hiding JavaScript using XHTML comments, the other using a CDATA section. Now, I have two issues that I would like you to analyse:

1. What should happen if you use just XHTML comments to hide JavaScript in an XHTML page?
2. Is the following code snippet correct?

```
<script type="text/javascript">
<![CDATA[
document.write("This is a test");
]]>
</script>
```

Comments

Answers and further explanations can be found on the following web pages:

http://www.w3.org/TR/xhtml1/#diffs (Section 4.8, 'Script and Style elements')

http://javascript.about.com/library/blxhtml.htm

http://en.wikipedia.org/wiki/CDATA

Literals

Programming requires manipulation of data. In JavaScript, data that appear directly in a program are called literals. For example:

```
"Hello world"
15
null
true
false
```

Variables and data types

A program normally handles data in some form or other. For example, a program created to calculate and print a credit-card bill would deal with the:

- amounts that have been charged to the card
- names of the companies charging the card
- dates of charges to the card
- card owner's name
- card owner's address.

All these items can be termed data, which can be processed by a program.

In this example not all the items are the same type of thing. The amounts charged to the card are all numbers, such as £2.99 or £45.00, whilst the names of the companies charging the card are all small pieces of text, such as ‘OU bookshop’ or ‘Smith and Bones Car Tyres Ltd’. The other distinct type of data is the dates. They could be seen as texts, for example ‘22 July 2003’, or they might be seen as a special ‘Date’ type of object which, unlike other text objects, can have many different values. For example, dates might have ‘short’ and ‘long’ forms with specific values such as ‘01/10/1999’ (‘short’ version) and ‘First of October Nineteen Ninety Nine’ (‘long’ version of same date).

The classification of elements into different types as I have started to outline is important because of what can be done with different types of value.

Take, as an example, the amounts charged to the credit card. These are numbers and they can be easily added to generate a total charge. The dates, on the other hand, cannot sensibly be added; the results would be meaningless. But, even with dates, some sensible mathematics can be performed. For example, the current date could be ‘subtracted’ from the date in the future when payment is required; this would give the number of days until a payment is required and the card owner could be told: ‘You have 10 days to make a card payment.’ It is difficult to see what ‘adding’ two texts would mean, but you might imagine that texts need to be joined (‘concatenated’) to form another, longer, text.

Each programming language comes with a set of ‘data types’ that can be used to store different types of data and which support a range of related operations.

JavaScript supports a number of different data types. The Core JavaScript Guide identifies the following:

- Numbers (integers and floating point)
- Strings
- Booleans
- Objects (including Arrays).

Within this Module, you can use this classification for the data types, but be aware that other JavaScript references (e.g. *JavaScript: The Definitive Guide* – see ‘Debugging JavaScript’ in this Study Guide) may identify or group the data types in a different way.

Activity 4 Data types

You should now read the explanation of the different data types in the Open JavaScript Guide, in the sections covering literals (‘Array literals’, ‘Boolean literals’, ‘Integers’, ‘Floating-Point Literals’, ‘Object literals’ and ‘String literals’).

After that, try to answer the following questions:

- What is the use of the special value NaN?
- Is ‘`Hello World`’ a valid value for a string?
- What about ‘`Hello\nworld`’?
- What are the essential differences between ‘primitive’ data types and objects?
- What distinguishes arrays from other objects? When would you use an array rather than a plain object to store and structure data?

Variables as space for data

You have now seen that different values are of different types. You may ask yourself how values are used and manipulated.

The answer to this starts with understanding how values are stored. To store a value in a program it is necessary to reserve space for it. Since the space required to store a value generally depends on its type, it is normally necessary to specify what type of value it will have. How specific you have to be about this depends on the programming language: some are fairly free or lax and have few storage types; others are very specific, have many types and are very strict in adhering to the types. JavaScript, for example, is fairly loose compared with languages such as C and Java.

The languages that have many types and are very strict about what can be stored in a space specified to be of a certain type are called 'strongly typed' languages; others are termed 'loosely typed'. There are also languages that have only a single type, which can be used to store any type of data.

So, why do we need to specify the type of data if it is possible to have a language that allows us to avoid this? The answer is in three parts. Firstly, typing is good in terms of correctness; strongly typed languages constrain the programmer and that alone helps to prevent errors. Secondly, a typed program can be much more closely checked for errors by the compiler or interpreter. If, for example, a date item is known to be a text item you cannot add another to it, and this type of error can be detected. Finally, a compiler can work out, for example, the maximum amount of storage space that a program will need for data and make sure that it has enough. Typed data thus makes memory management easier.

Declaring variables and storing values

If a program is needed to perform some useful task, it is normally necessary to use a range of values. Suppose, for example, a program maintains the balance of your bank account. This program receives a stream of numbers that represent money coming in and going out of your account. If the program is only to manage the next payment in or out of the account and to keep the balance of the account then it needs to deal with only two numbers at any one time.

Suppose that this is the case, and you have an account that starts with £1000 in credit. In the program, this value needs to be stored as the current balance and then added to, or subtracted from when you put money in or take money out.

To store a value some space is required (actually an area in the computer's memory). The value is a number, so at the very start of the program the computer has to be told that a number is to be stored. To do that, some way of referring to the number is needed; for example the name 'accountBalance'. In this case the number will start off with a value of £1000.

In a program this might look something like:

> 'accountBalance' is a number with initial value £1000.

But that's really quite a complex English language statement for a computer to understand.

Instead, in a programming language (but not in JavaScript!), it might be something like:

```
number accountBalance = 1000;
```

This is quite often termed a program **statement**; there are several kinds of these, as you will see. This particular statement has two effects. Firstly it says that the variable accountBalance is a number, that is, it says what type of variable it is. This is called **typing**, and statements that do that are called 'type statements'. Second, the statement gives the variable a value using '='. The '=' operator is typically used to give values to variables, although one or two languages use a different symbol. Giving a variable a value in this way is called **assignment**, so this is also an 'assignment statement'. The first time you assign a value to a variable is typically called **initialisation**.

Statements that declare a variable's name and type are also frequently called **declarations**. A variable normally has to be declared before it can be given a value (as above) or used in any other way.

In JavaScript, you declare variables using the `var` keyword:

```
var x;
```

You can even assign a value to a variable to initialise it while declaring it:

```
var x = "Hello world";
```

and you can combine multiple variable declarations and initialisations in one statement:

```
var x = "Hello", y = " ", z = "world";
```

Expressions

Programming in any language requires manipulating data. In some cases, the data will be just literals (such as text strings) but quite often more complex **expressions** will be used.

Anything that can be evaluated by the JavaScript interpreter can be thought of as an expression, just as literals are any data that appear directly in a program.

Expressions can use a number of different operators, or no operators at all.

For example:

```
x * 2
x + y/4
x - y + 2
```

Operators

In JavaScript, operators are special functions that operate on one to three values (operands).

The most common operators require one operand (unary operators) or two operands (binary operators).

Unary operators require a single operand, which can be before or after the operator. Binary operators require two operands, one before and one after the operator.

By now you have seen a few operators that are a part of JavaScript, for example the assignment operator ('`=`') and the operators for the basic math operations (addition, subtraction, multiplication, division).

JavaScript includes a good number of operators (more than 30), but here we are going to focus on those that are most common and more useful for your forthcoming studies.

Assignment operators

Assignment operators are used to assign the value of the right operand to the left operand:

```
x = y;
```

The right operator can be any type of expression:

```
x = x + y;
```

Beside the 'plain' assignment operator, JavaScript includes a number of shorthand assignment operators:

```
x+=y;
```

is equivalent to:

```
x = x + y;
```

JavaScript includes shorthand assignment operators for the four main arithmetic operations: +=, -=, /=, *=, plus a number of others.

Arithmetic operators

JavaScript's simplest arithmetic operators are: addition (+), subtraction (-), multiplication (*), and division (/).

Other useful arithmetic operators are the unary (with just one operand), increment (++), and decrement (--) operators.

For example:

```
x = 1;
x++; // x is set to 2
y = x++; // y is set to 2 and x is set to 3
z = ++y; // y and z set to 3
```

I will discuss operator precedence later.

Comparison operators

Comparison operators are used to compare the operands and return a Boolean value (True or False).

Operator	**Symbol**
Equal	==
Strictly equal	===
Not equal	!=
Strictly not equal	!==
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=

The equal and not equal operators require some further comments. First of all, remember that there is an operator for assignment (=) and an operator to compare if two variables are equal (==). The strictly equal and strictly not equal operators compare the type of the variable too – if the type does not match, the variables are not equal.

Logical operators

Logical operators are used to connect different expressions. Logical operators may require that two expressions are both true, that at least one is true, or can be used to negate an expression.

The basic operators are and (&&), or (||), not (!).

Look at the examples:

```
x == 1 && y ==2    (evaluates to True if x equals 1 and y equals 2)
x == 1 || y == 2   (evaluates to True if x equals 1 or if y equals 2)
x != 1             (evaluates to True if x is not equal to 1)
```

String operators

JavaScript has an operator (+) to concatenate strings. We have already encountered the + operator, used to add two numbers. You can see how the + operator has two different meanings, one between numbers and another between strings.

Moreover, the + operator can be used to transform a number to a string, concatenating an empty string to a number:

```
x = 1 + 3;
x += ""
```

Activity 5 Play with a longer script

Writing JavaScript code from scratch can be challenging and it may take some time before you are able to write code that is either useful or at least amusing. As an early step, set up the following code in your chosen code editor and play with it:

```
<?xml version="1.0">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
    <title>Example</title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1"/>
    </head>
    <body>
        <script type="text/javascript">
        /* <![CDATA[ */

    var haveCar = true;  // a boolean  type
    var sequence = 0;   // a number type
    var message = "Glad that you have not disabled your javascript";
    // a string type
    var lights = [ "red", "amber", "green"];  // an array object
    var actions = { red : "stop", green : "go", amber : "caution"};
    // a plain object

    // comment: now, off we go
    alert( message );

    if ( haveCar == true )  {
    document.write( "The lights are " + lights[ sequence ] );
    document.write( " which means " + actions[ lights[sequence] ] );
    }  else {
    document.write( "you have no car" );
    }
    /* ]]> */
    </script>
    <noscript><p>You have disabled javascript</p></noscript>
</body>
</html>
```

See what happens when you:

- disable javascript in your browser
- change the value of the number: what happens if it is more than 3? Why might that be?
- change the boolean from true to false, or give it the numeric value 1 or 0
- add another colour or an action to the array or the object?

You can't damage anything by playing with code in your code editor and browser. If your code contains errors or gets stuck in an endless loop it will either not run, or run until Firefox decides to end it.

Activity 6 Evaluating expressions

Try to find what these expressions evaluate to. First, try without running the code and then try to test the code and find the value.

```
1 + 5 == 3 * 4 - 2 * 3
5/2 > 2.49
3 === 2 + 1
4 === '4'
'5' == 4 + 1 + ''
51 === ' ' + 5 + 1 - 0
51 === 5 + 1 + ' ' - 0
```

To test the value of an expression, just alert() the value of the expression:

```
alert(3 === 2 + 1);
```

You will note some surprising results. The evaluation compare the values that result from the right hand side of the expression with the value of the left hand side of the expression. In some cases, the comparison asks for comparison of type as well as value. JavaScript makes intelligent assumptions about type as it reads the expression from left to right, and its assumptions can change the resulting value from what you expected.

No coder would write these expressions in real life, but when writing code, you need to take care that JavaScript's assumptions about type are those you intended.

Comments

If you want to read more about JavaScript operators, have a look at the documentation in the Core JavaScript Guide, and more specifically the sections ‘Operators’, ‘Assignment Operators’, ‘Comparison Operators’, ‘Logical Operators’, ‘String Operators’ and ‘Special Operators’.

Operator precedence

You will have seen in the examples in the previous pages that there are situations in which different operators are used, one after the other. If an expression has more than one operator there are multiple ways in which the expression can be evaluated. The expression:

```
4 / 2 + 2 * 3
```

could evaluate to 12 or to 8 depending on which operator is applied first. To prevent this type of ambiguity, operators are each given a **precedence** and an **associativity**.

Precedence controls the order in which operators are evaluated; a higher precedence operator, a larger precedence value, is evaluated first.

Multiplication and division have a higher precedence than addition and subtraction. They will therefore be evaluated first, in our example expression.

Associativity describes how to process them (right-to-left or left-to-right). Associativity for arithmetic operators is left-to-right, and for comparison operators is right-to-left.

Now, if we have a look at this code:

```
var x = 1;
var y = 2;
```

```
var z = 3;
y += x *= z;
```

The value of x, y and z after the last operation depends on whether `x *= z` or `y += x` is executed first. As we have two assignment operators, and their associativity is right-to-left, we know what will be the order in which they will be analysed.

One good practice is simply to bracket expressions to make it clear how the expression should be evaluated. In this case you do not have to know the precedence or associativity of any operators. The statement given above can be bracketed to make the two evaluations clear.

```
y = ((y + x) * z) // y = ((2 + 1) * 3), giving y = 9
y = (y + (x * z)) // y = (2 + (1 * 3)), giving y = 5
                  // y += x *= z is equivalent to y = (y + (x * z))
```

A more readable way to express the same code would be:

```
x = x * z;
y = y + x;
```

For a table with the precedence of the other operators and associativity, you should check the Core JavaScript Reference:
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Operators:Operator_Precedence

Operators and type conversions

We have already looked at the different data types in the previous pages.

JavaScript is generally labelled a ‘loosely typed’ language because variables are just named and used, rather than named and given a type. JavaScript has a flexible treatment of variables, but this has its limits. For example, it is possible to compare two strings or two numbers to see if they are equal, but what happens if a number is tested to see if it is greater than a string?

The answer is that JavaScript does its best to convert the values involved so that the operation can be evaluated. This can lead to some unexpected results, as JavaScript shows no favours, neither to integers nor to Booleans; it’s every type for itself.

Conversions can be demonstrated by a script that creates variables holding different types of value and then prints these out as if they were strings. Below is a script snippet you can embed in a web page that does this. You might like to know that ‘new Number(age)’ creates a new object of type ‘Number’ which has a value ‘40’, because ‘age’ itself is a variable.

```
<script type="text/javascript">
/* <![CDATA[ */
    var age = "40";
    document.write("my age was: "+ age +"<br />");

    var newAge = age + 1;
    document.write("my age is now: " + newAge + "<br />");

    var myNewAge = 1 + age;
    document.write("my age is now: " + myNewAge + "<br />");

    var myOldAge = new Number(age);
    var myVeryNewAge = myOldAge + 1;
    document.write("my age is now: " + myVeryNewAge + "<br />");
/* ]]> */
</script>
```

Debugging JavaScript

A very simple approach to debugging your JavaScript is to add debugging statements in JavaScript to your code. For example to see the value of a variable it can be included in an 'alert' which pops up a dialog box which can display the variable' value.

JavaScript code can be complex, and so it is important that one of the skills you acquire is to be able to identify bugs in the code (this is called debugging).

Activity 7 Simple script debugging

To see how this can work try including the following g script in an HTML page.

```
<script type="text/javascript">
  var number = 21;
  alert("number is " + number);
  var another_number = 23;
  alert("another_number is " + another_number);
  var times = 21 * 23 ;
  alert("times is " + times);
  var final = 6 * times ;
  alert("final is " + final);
</script>
```

As the script is executed line by line each variable is given a value and then an alert displays that value. The script doesn't continue onto the next line until you dismiss the alert by clicking the 'okay' button in the alert.

This is also useful if you have a syntax error in a script as when the script reaches the error it will cease to execute and the last alert message will identify where the error lies. Try this by adding 'zzzz' to the sixth line:

```
var times = 21 * 23 ; zzzz
```

The last alert you will see then will be the one on line 5. So the error must lie between the end of line 5 and the end of line 7 otherwise the alert on line 7 would be seen.

Try now to inspect the following code.

```
<script type="text/javascript">
/* <![CDATA[ */
var hello_world = "Hello World;
var text_to_alert = "";
for (var c=1; c<=5; c++) {
    text_to_alert += hello_word + ": " + c + " time.\n";
}
alert(text_to_alert);
/* ]]> */
</script>
```

There are two errors in the code; can you see them? If you can see the errors – well done! It means that you are getting familiar with JavaScript syntax.

Don't worry if you can't see the errors, you will can learn how to find errors using Firefox.

If you have not done it yet, this is the time to download and install Firefox (http://www.mozilla.com/firefox).

You can use Firefox to debug your code.

Firefox includes a JavaScript console that allows you to see the errors in your scripts and to run JavaScript commands in a command line. To activate the JavaScript console in Firefox, click on Tools > Error Console. You will see something similar to Figure 2.

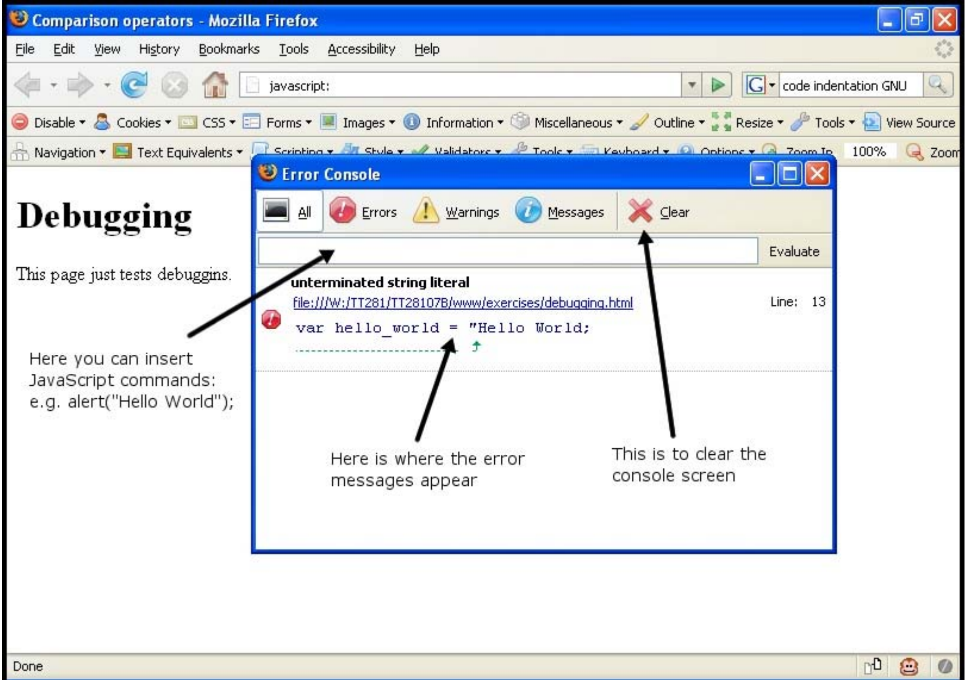

Figure 2 Using Firefox for debugging

Besides the JavaScript console, there are other tools that can be useful:

- JavaScript lint, http://www.javascriptlint.com
- JSLint, http://www.jslint.com
- Venkam JavaScript Debugger, http://www.hacksrus.com/~ginda/venkman
- Firebug, http://getfirebug.com

Activity 8 Firefox debugging

Use the JavaScript console of Firefox or any of the other debugging tools to try to find the errors in the script above. Try to familiarise yourself with the tools. Which one or which ones do you think is more useful? Why?

If you didn't manage to complete any of the previous JavaScript activities, this is a good moment to have a second look at them, using these new tools to try to find your errors.

Statements

Statements are JavaScript instructions that tell the interpreter to perform an action. Some examples are:

```
x = 1;
counter --;
alert(3 === 2 + 1);
```

These types of statements are typically called *expression statements*, and are the simplest type of statements available in JavaScript. Expression statements include assignments, method calls, increments and decrements.

Activity 9 Read about statements

You should now read the section 'Statements' in the Open JavaScript Guide, excluding the sections 'Exception Types', '`throw` Statement' and '`try` ... `catch` Statement'.

The following sections of this guide are intended to provide you with material to supplement the Open JavaScript Guide explanations and examples. The guide should be particularly useful if you are fairly new to programming.

Conditional statements

The `if` statement allows you to test an expression and then make a *branch* in the script's flow of control. You can nest `if` statements, obtaining multi-way branches: the branch that will be followed during the execution will depend on the value of the expression that is evaluated in the `if` statement.

If you look at the following example, there are three different branches, and the branch that will be followed during the execution depends on the value of `number`:

A common mistake is to use an assignment ('=') when comparison is intended ('==')

```
if (number == 1) {
    // first branch:
    // some statements applicable if number is '1'
} else {
    if (number == 2) {
        // second branch:
        // some statements applicable if number is '2'
    } else {
            // optional 'catch all' branch
            // statements applicable if number
            // is not '1' or '2'
    }
}
```

The `if` example above takes the approach of repeatedly testing the value of `number`. However, if all the branches depend on the same expression (in this case the value of `number`) it is more efficient and clearer to use a `switch` statement. Here is the above nested `if` statement rewritten as a `switch` statement:

```
switch(number) {
    case 1:

        // some statements applicable if number is '1'

        break;
    case 2:
        // some statements applicable if number is '2'

        break;
    default:
        // optional 'catch all' branch
        break;
}
```

The execution of the `switch` statement is quite complicated. Depending on the value of `number` the first matching case branch is selected. The statements for that branch are then executed (I have just included a comment). The `break` command then causes execution to break out of the switch statement and continue. If the `break` is not included at the end of the selected case then execution continues within the switch statement and further cases may be executed. It is more usual to include the `break`

statement in every case. If none of the cases are executed then the optional `default` case is executed.

Loop statements

One of the more used loop statements is the `for` statement. It is used to perform a sequence of one or more instructions a number of times.

The general form of the `for` loop is:

```
for ( [initialisation]; [test]; [increment] ) {
     [statement]
}
```

The loop typically initialises a counter which is then tested against a terminating condition. If the condition is true the statement is then executed and the counter is incremented before repeating the process. The simple example below will output the numbers 0 to 9, one number per line:

```
for (var counter = 0; counter < 10; counter++) {
     // we are concatenating the value of counter with
     // an XHTML break line, <br />
     document.write(counter + "<br />");
}
```

Please note that there may be situations in which you do not need all the elements of the `for` loop (initialisation, test, increment, statement).

For example, this is a valid `for` loop:

```
for (var counter = 0; counter < 10;) {
     alert(counter++);
}
```

Another loop statement is the `while` loop, which has the form:

```
while ( [expression] ) {
     [statement]
}
```

The previous `for` loop example can be recast as a `while` loop:

```
var counter = 0;
while (counter < 10) {
     document.write(counter + "<br />");
     counter++;
}
```

Notice how the initialisation takes place before the loop statement occurs, and how the expression is altered (incremented in this case) inside the loop. If the counter were not initialised first it could not be tested against '10' and you would see an error. If the expression were not changed inside the loop it would never come to be evaluated at anything but 'true', so you would have an infinite loop.

This is not a very good use of a `while` loop, and in this case the original `for` loop is better. If the number of iterations for the loop is not already known then a `while` loop is typically more appropriate.

Activity 10 Looping

a. Consider the 'while' loop in the following code:

```
var countdown = 10;
while(countdown > 0) {
      document.write(countdown+ '...');
      countdown --;  // alternatively: countdown = countdown -
```

```
1;
};
document.write('Bang!');
```

Now, write that using a 'for' loop to get the identical output.

b. JavaScript has a 'prompt' function that displays a box for a visitor to enter information. Optionally the scripter can include a message in the prompt box. When the visitor enters input and clicks 'OK' the input can be stored in a variable for later use by the script. If the visitor clicks 'Cancel' the prompt returns the value **null** to be stored in the variable.

Look up the prompt box in the JavaScript guide at https://developer.mozilla.org/En/DOM/Window.prompt

Write a short script that declares a variable named input and asks for input from the visitor in a prompt box. Repeat this prompt in a while loop that prints to the page the text that the visitor entered, unless the visitor clicks the 'Cancel' button, whereupon the loop exits with a closing message 'Goodbye'. We can write the logic of this as follows:

// declare variable to store prompt('Enter some text or click Cancel to finish',"")

// while the value of input is not null

// write the input text to the page plus an HTML linebreak

// prompt for more text to be input

// the loop will exit if the value of input is null, whereupon write 'Goodbye'

Place your script in script tags in the <body> (not the <head>) section of the document. Test it.

What happens if you prompt for more text within the loop before you write the value of input to the page? (Think, and test).

Arrays

Earlier in this guide you encountered some of the data types that you can use in JavaScript: Number, String and Boolean.

I'd now like to introduce you to arrays, which are a type of object (we will explore objects later), that is typically used to store a number of related items.

The array is a very significant type of object. Found in most programming languages, it is used to store a set of values. An array can be pictured as a collection of variables with the same name (the name of the array) and a series of numbers by which they are indexed.

0	1	2	3	4	5
12	230	233	1	100	0

Figure 1 An array structure

The array in Figure 1 represents the number of votes in an election for six different candidates. Thus, it has six elements numbered 0 to 5 (indexing always starts at 0) and six values which have been assigned to each element (which in our case represents the number of votes).

If the array was called `myArray`, it might have been created using:

```
var myArray = new Array(6);
myArray[0] = 12;
myArray[1] = 230;
myArray[2] = myArray[1] + 3;
myArray[3] = 1;
```

```
myArray[4] = 100;
myArray[5] = 0;
```

Here, `new` is the JavaScript operator to generate a new instance of any object type and **6** is the array length. If we didn't know we had six elements, we could have used the *generic constructor* – the same as above but without a specified number of elements – to make a new array:

```
var myArray = new Array();
```

If values are known at the time of creating an array, they can be assigned to the elements in a single statement:

```
var myArray = new Array(12,230,233,1,100,0);
```

Once the array is created, any elements can be set or retrieved. To refer to an element rather than the entire array, the *array name* together with the *element number* enclosed in square brackets is used.

For example:

```
myArray[3] = 99;    // set 4th element to 99
```

This will change the array, as shown in Figure 2.

0	1	2	3	4	5
12	230	233	99	100	0

Figure 2 Modified array structure

Similarly, values from the array can be incremented and printed out:

```
myArray[3] = myArray[3] + 1;     // increment 4th element to 100
document.write("value in cell 4: " + myArray[3]);
```

Working with arrays

You have already met a range of 'operators', that can be used to make new values, 'assignments', for giving variables values, and 'comparisons', that can be used to construct expressions. You should have already completed a few exercises involving expressions in Study Guide 4. However, since arrays introduce additional complexities, this section provides further material to help ensure that you are comfortable with their use.

One very common way of using an array involves using a `for` loop to access every element (or a range of elements), as the loop's number can be used as the array element number. The code below prints out every element of the array above.

```
for (var i = 0; i < 6; i++) {
    document.write("Votes for candidate " + i +
        " = " + myArray[i]);
}
```

This example demonstrates how a data structure such as an array can be efficiently handled by certain control statements such as a loop.

Activity 11 Testing an expression

The following script evaluates an expression and then puts up an alert box stating whether the expression is true or not:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Expression Testing</title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1"/>
```

```
</head>
<body>

    <h1>Testing an expression.</h1>

    <script type="text/javascript">
/* <![CDATA[ */

    // saving an expression as string in expr
    var expr = "3 < 2";
    // evaluating the expression
    if(eval(expr)) {
        alert("The expression is true!");
    } else {
        alert("The expression is false!");
    }

    /* ]]> */
    </script>

</body>
</html>
```

In this script we are introducing a new function, `eval`. This enables you to store expressions in a variable and to evaluate them a second time.

You can try out the script as it is, and then modify it to test other expressions. For example, try to work out what the following expressions evaluate to and then check your answer using the script or a modified version of the script.

```
((3 != (2 * 2)) && (2 >= 1))
! ((4 + 2 + 3) > 9)
x = 3
(2 || 4)
```

The next step is to store all the strings to evaluate in an array, and to evaluate them with a loop.

Comments

In this activity you encountered `eval` for the first time. Note that `eval` is quite complex – don't feel demoralised if you don't manage to solve this activity immediately.

Moreover, the last two expressions are especially peculiar. They do not appear to have Boolean (true or false) solutions. The third is an assignment, setting one thing equal to something else. Can an assignment be anything but true? Booleans are defined as being either `FALSE`, which is often a synonym for zero, or `TRUE`, which can be represented by any other value.

Objects

Objects are a collection of properties, each of which has a name and a value. A property of an object is simply an additional piece of information held in the object about the object.

Objects are typically used when you need to structure information:

```
var john_wilson = {"Name": "John",
        "Surname": "Wilson",
        "OUCU": "jw9372",
```

```
        "PI": " A1234567"}
john_wilson.Region = 1;
```

To access the value of a property, you have to use a 'dot' notation, as you can see from the last line of the example. An alternative syntax is to use the `[]` operator to access properties – as with arrays:

```
john_wilson["Region"] = 2;
```

You can use a `for` loop to retrieve all the values of the properties of an object (and in this case you have to use []):

```
for ( var i in john_wilson ) {
alert(i + ": " + john_wilson[i]);
}
```

Before you progress any further you should read a few sections from the Open JavaScript Guide. In the chapter 'Working With Objects', you should read 'Objects and Properties', 'Creating New Objects' and 'Predefined Core Objects' (excluding the section 'RegExp' Object).

The Open JavaScript Guide covers these topics more in depth than the Study Guides and will give you additional examples.

Functions

As you may have already seen, you do not need to write all your JavaScript code or be familiar with all the JavaScript code that you are using. To use a function, you only need to know the parameters it requires and the outcome of the function.

This is how you write a function:

```
function nameOfTheFunction([parameter_1], [parameter_2], [parameter_3]) {
    // you can use parameter_1, parameter_2, parameter_3 in your statements
    // parameter_1, parameter_2, parameter_3 are local variables, valid only inside
the function
[statements]
}
```

and this is how to invoke (or 'call') it:

```
nameOfTheFunction([parameter_a], [parameter_b], [parameter_c]);
```

The value of `parameter_a` will be passed to `parameter_1`, the value of `parameter_b` will be passed to `parameter_2` and the value of `parameter_c` will be passed to `parameter_3`.

`parameter_a`, `parameter_b`, `parameter_c` are said to be global variables, while `parameter_1`, `parameter_2`, `parameter_3` are called local variables, valid only inside the function.

The outcome of a function can be a value returned explicitly by the function. As a simple example consider the function called 'product' below which takes two parameters, multiples them together and then returns the result.

```
function product(number1, number2)
{
        return number1 * number2;
}
```

This function can be called, for example in the body of an HTML page:

```
document.write(product(4,3));
```

which will write the result '7' returned by product.

Writing functions

The next step is to write your first function. The first functions you will see here may be challenging, but later, when you have completed reading, it should be easy to identify functions and to understand (at least from a general point of view) what they do.

Below is a first example of a function, together with the enclosing XHTML and code invoking the function:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Functions</title>
<meta http-equiv="content-type" content="text/html;
charset=iso-8859-1"/>
<script type="text/javascript">
/* <![CDATA[ */
    function addNumber(number1, number2) {
        var number3 = parseInt(number1) + parseInt(number2);

        // the next line returns the value of total back
        // to the global context
        // otherwise it would not be available any more
        return number3;
    }

/* ]]> */
</script>
</head>
<body>

<script type="text/javascript">

var total = 0;
var number = prompt("Add a list of numbers. Type a number or '.' to exit.","");

while(number!=".") {
    total = addNumber(number,total);
    number = prompt("Add a list of numbers. Type a number or '.' to exit.","");
}

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

</script>

<p>This script just shows how to use functions.</p>
</body>
</html>
```

The script asks the user for a number or a full stop to terminate the execution. It saves the number entered by the user in the variable `number`, and then it passes the value of `number` and `total` to the function.

Inside the function, the value of `number` is copied to `number1`, and the value of `total` is copied to `number2`. Changing `number1`, `number2` or `number3` does not affect the main context – that's why we have to return `number3` and assign the value returned by the function to `total`.

This mechanism, whereby a value is copied and passed to a function, is called ‘parameter passing by value’.

Activity 1

Try to run the code of the JavaScript example above. Then, modify the example moving the while loop inside the function and modifying the rest of the script accordingly.

Which solution do you think is better? Why?

Variable scope

The scope of a variable is the region of code where it is defined and may be used. A 'global' variable has global scope: it can be accessed anywhere in a script. Other variables such as those declared within a function are 'local' variables and may only be accessed within the function. Local variable names take precedence over those with greater scope.

Activity 2

Read the following code and try to determine what should happen. Then run the code and see what actually happens.

Functions allow you to reuse code effectively, without having to duplicate it.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Function example</title>
<script type="text/javascript">
/* <![CDATA[ */
    var number = 200;
    incNumber(number);
    alert("The first value of number is: " + number);

    function incNumber(number) {
        // what is the value of number here?
        number++;
    }
    // what is the value of number here?
    number++;
    alert("The second value of number is: " + number);
    incNumber(number);
    // what is the value of number here?
    alert("The third value of number is: " + number);
/* ]]> */
</script>
</head>
<body>
<p>This is just an example of a function.</p>
</body>
</html>
```

Comments

In general, you can think of scope in terms of localities; your street, your village/town, your county, your country, etc. If someone mentions 'Fred Smith' that's generally taken to mean the name of someone in the locality, even if there are people with the same name further afield.

Constructors

Constructors are a special type of function that can be used to create many instances within a group of similar objects that share properties and methods. Only the values of those properties will differ from one instance to another.

Earlier we saw how we can create an object in this way:

```
var john_wilson = {name : "John",
                   surname : "Wilson",
                   oucu : "jw9372",
                   identifier : "A1234567",
                   region : 1
                   };
```

If we have a range of students sharing these properties then we can both save time and ensure they are consistent in form by using a constructor function. You have already come across constructor functions if you have used the expressions **new Array()** or **new Object()**. Thus Array and Object are the names of the constructors, and they must be called with the **new** operator. The new operator creates an empty object and supplies it to the constructor for shaping. Inside the constructor, the value of **this** is the new object. When we write the constructor function, we haven't got an object to work with, so we write it essentially as a template that will work on '**this**' new object when the new object is supplied in the function call.

If you forget to call a constructor function with the **new** operator, it won't have a new object to work on and the value of **this** will be the global object. This is bad. To help the programmer identify constructors in a script, they are by important convention given a capital initial letter.

Let's make a Student constructor:

```
function Student(name, surname, oucu, identifier, region) {
        this.name = name;
        this.surname = surname;
        this.oucu = oucu;
        this.identifier = identifier;
        this.region = region;

};
```

Now we'll use it to make an individual student record:

```
var john_wilson = new Student("John", "Wilson", "jw9372", "A1234567", 1);
```

We can now access the properties of john_wilson using either dot notation or object 'array' notation:

```
alert(john_wilson.oucu);
alert(john_wilson["name"]);
```

You can make any number of individual students using this paradigm.

Activity 12 Constructing objects

Set up the Student constructor in your code editor and make sure you can call it with different sets of arguments for two students, and then access their properties. Use a for...in loop to loop through the properties of a given student and output the results to the page.

Methods

Methods are functions bound to objects just as static properties are. When a method that is bound to an object is invoked (called), the value of **this** within the method is the parent object.

Let’s add a parameter for the student's course, and methods for praise of and complaint about the course. We can add a method either by defining a function within the constructor, then binding it as a property of **this;** or using a single step with a function literal. You will see each variant here:

```
function Student(name, surname, oucu, identifier, region, course) {
        this.name = name;
        this.surname = surname;
        this.oucu = oucu;
        this.identifier = identifier;
        this.region = region;
        this.course = course;

        // methods: first the literal
        this.praise = function() {
                        return this.course + ' is awesome !';
                    };

        // now the two-stage version
         this.grumble = grumble;

              function grumble() {
                return this.course + ' is far too hard  grumble grumble';
           };
 };
// make a new student
var john_wilson = new Student('John', 'Wilson', 'jw1234', 'A567890', 1, 'TT281');
```

We call the methods with the call operator() as usual.

```
// hear him praise his course:
alert ( john_wilson.praise() );
// and complain
alert ( john_wilson.grumble() );
```

Activity 13 Using a constructor

Add a property to the constructor to store a further parameter, the student's year of birth (a number, e.g. 1973). Add a method to calculate the student's age in years during the current year. You will need to look up the Date constructor and its methods in the Core JavaScript Guide. First, set up a separate script to make sure you can get the current year from a new Date(), subtract the year of birth and return the age difference. Make that into a function and transplant it within the Student constructor as a method. Make a new student by calling the constructor (remembering to add their year of birth to the function call). Test your method.

Activity 14 Further reading on functions, objects and methods

You may now find it useful to read a few sections from the *Open JavaScript Guide.*

In the chapter ‘Functions’, you should read the sections ‘Defining Functions’, ‘Calling Functions’ and ‘Using the arguments object’. If you have not previously done so also read the sections on constructors and methods.

It may then be useful to reread the sections 'Using a Constructor Function', 'Defining Methods' and 'Using this for Object References' in the chapter 'Working with Objects'. Note that at some points the Guide errs in not writing constructor functions with an initial capital letter.

Coding guidelines

This section provides some guidance for coding and, in particular, producing code that is clear and well structured, and hence easier to debug and maintain. If you have no or very little experience in scripting you should read this section. The guidelines are very basic, but cover essential elements for good programming practice.

Why do we have coding guidelines?

In the early years of computing, computer resources were severely limited and many of the tasks for which computers were used were very mathematical in nature. Mathematicians use very concise notation ($a = b + c$ is probably not regarded as beautiful poetry, unlike Shakespeare's 'Shall I compare thee to a summer's day?', although some mathematicians might disagree) and they carried this over to their writing of programs. In the early days of programming, a single individual was often the sole author, maintainer and user of the programs he or she used.

Today, a complete application can, and typically will, involve many hundreds of thousands of lines of code. This code will have been written by a substantial team and may include libraries and code written elsewhere some years ago. The code will also embody a great deal of knowledge of the precise application implemented, not to mention the investment involved in its production.

So, when something goes wrong or when something needs to be altered or upgraded it is very important that the code can be easily understood, perhaps by someone not involved in the original project. That person might be you. Alternatively, even if it was you who wrote the code two years ago, will you be able to understand it now?

There are a number of simple measures you can take to give your code 'quality'. This will allow you and others to debug, maintain and extend your code far more easily.

Indentation style and layout

If you make good use of white space and indentation your code will be much easier to read and maintain. It is usual to insert a tab or some indentation spaces (typically 2 or 4) when you start a new block statement and to align the opening and closing symbols for blocks.

For example:

```
if (name == "heap")
{
    lecturer = true;
    forename = "nick";
}
else
{
    lecturer = false;
    forename = "";
}
```

An alternative, more compact, formatting style is to keep the opening bracket in the first line of the code block:

```
if (name == "heap") {
```

```
    lecturer = true;
    forename = "nick";
} else {
    lecturer = false;
    forename = "";
}
```

Some editing tools provide automatic formatting and allow customisation of the formatting they perform. The important point is to adopt a formatting style that is readable, consistent and that supports editing of statements in a reasonable fashion. In the ‘if’ statement above it is a simple matter to delete the ‘else’ part or to add another ‘else’ clause.

Observe the indentation and layout used in the Study Guides, the ebook, and other JavaScript code you encounter. Assess how easy it is to read those examples compared with your own efforts, then start practising in your own scripts.

Using spaces or tabs is typically a matter of preference, but try to be consistent in your scripts, using either spaces or tabs.

Be aware that in some cases you will have to adapt your style preferences to comply with given guidelines. This may be because of a request from a customer, or it may be the case if you want to have your code accepted in other projects.

‘Optional’ semicolons and ‘var’s

JavaScript, in addition to being indifferent to white spacing and indentation, is very forgiving and will make certain assumptions for you.

The first assumption is about what constitutes a JavaScript statement. As long as each JavaScript statement occurs on its own line, the JavaScript interpreter will assume that the line constitutes a single JavaScript statement. Other languages aren't that flexible, requiring that every statement be separated from every other statement by a delimiter. JavaScript's ‘optional’ ending delimiter of a semicolon is required by many other languages, including PHP, Java, Perl and C. It is therefore a good idea to get into the habit of using them.

The second assumption JavaScript makes for you is that a variable declared without a ‘var’ is global in scope, accessible and changeable from everywhere. This approach to declaring variables violates good programming practice and isn't conducive to proper object-oriented design. What we want to do is pass needed information into functions as parameters and return modified values, as we did earlier when we examined variable scope.

The JavaScript ‘var’ keyword forces the JavaScript interpreter to treat the variable declared as local to its region. Normally, a region, or block, in JavaScript is any chunk enclosed by braces, i.e. ‘{’ and ‘}’ characters. A function is therefore a block but we also saw blocks for ‘for’ loops and ‘if’ statements. By using ‘var’ appropriately in your blocks and avoiding global variables where possible, you will find it easier to debug your code and to work towards developing reusable, modular code.

Consistent and mnemonic naming

For a mathematician ‘x’ and ‘y’ have specific meanings, but your programs will probably deal with more familiar concepts. Below are some variable names: which do you think carry meaning?

```
X
Variable1
US
String20
UserSurname
```

Suppose your code reads in a user's password and then tests to see if the user is claiming to be a person called Nick Heap. Somewhere in the code you will have read in the string the user has entered as his surname. Then you might write:

```
if (X == "Heap") {
    [statement]
}

Or you could write:
if (UserSurname == "Heap") {
    [statement]
}
```

This second version seems much clearer. Throughout the code you can see the `UserSurname` variable and know its intended use. You might also decide to use a name that includes the variable type (in JavaScript where types are quite free), so you might use `UserSurnameString`.

A common mistake is to forget that JavaScript identifiers are case-sensitive, so myName and myname are different objects.

An alternative standard is to separate keywords with underscores: so instead of `UserSurname` you would call the variable `user_surname`.

I can't recommend a specific standard: the key is to choose something obvious and to use a consistent naming and capitalisation convention for your variable and function identifiers (names).

Make appropriate comments

There are some situations in which you should use comments.

A file that contains several functions should describe what the functions are used for in general.

```
/*
   Functions for parsing strings. Functions here parse
   strings and perform common operations: changing
   case 'toUpper', 'toLower' and 'trim' removing spaces.
*/
```

You should also add comments to each function, especially any assumptions you have made in its writing. You should also comment on any specific values or special lines of code.

Reusable code

In the analysis of a problem, similar tasks are often repeated. This is the basic reason for writing functions. Instead of writing exactly or nearly the same code several times, the code can be encapsulated in a function and you can then simply invoke or 'call' the function as many times as required.

If there are only minor changes in the code, you may manage the differences using parameters in the functions, eventually using `if/else` statements in the function body. You have seen a simple case of this in the message function that generates an alert. If you have code that generates two different alert messages,

```
alert("This is a warning that time is running out");
....
alert("This is a warning that time has run out");
....
```

you can factor out the common parts to make up a function and then pass in the changes to the message as a parameter to the function.

```
function warnAlert(message) {
    alert("This is a warning that " + message);
}
```

```
....
warnAlert("time is running out");
....
warnAlert("time has run out");
....
```

Functions are a very good approach when there is a need for multiple occurrences of the same code. The code only has to be written and documented once, it only has to be debugged and edited once, and any future changes or enhancements only have to be made once.

One element of large-scale programming that you might like to adopt is to compile a library of JavaScript functions that you can reuse. There are many approaches to this; one is to use XHTML pages that both display the code, ready for cut and paste, and allow you to run the example to see how it performs.

To conclude, remember that it may often be the case when you are developing any project, that it is easier to explore ways of using already existing code (for example from Open Source projects). Using existing code can allow a faster development process. DO be sure to acknowledge the sources of any material you reuse.

Other sources of information

As long as you are a student at the Open University, you have access to Safari (**http://proquestcombo.safaribooksonline.com.libezproxy.open.ac.uk/home**).

If you want to further improve your knowledge of JavaScript, you can read the following chapters of *JavaScript, The Definitive Guide*: 'Functions' and 'Classes, Constructors, and Prototypes' (**http://proquestcombo.safaribooksonline.com.libezproxy.open.ac.uk/0596101996**).

The *Definitive Guide* is quite complex (and comprehensive). An alternative, lighter read would be the chapter 'Functions' in *Learning JavaScript*, again available on Safari (**http://proquestcombo.safaribooksonline.com.libezproxy.open.ac.uk/0596527462**)

From *Learning JavaScript* you can go on to read the following chapters: 'JavaScript Data Types and Variables', 'Operators and Statements', 'The JavaScript Objects'.

Similarly, if you want to explore JavaScript in more depth, I would recommend that you read the whole Open JavaScript Guide, not just part mentioned here

http://sourceforge.net/projects/javascriptguide/

If you want a reference text that covers all of JavaScript in great depth I can recommend *JavaScript: The Definitive Guide*, 5th edition, by David Flanagan. Published by O'Reilly, in 2006 (ISBN 0596101996).

The guide is also available in Safari: **http://proquestcombo.safaribooksonline.com.libezproxy.open.ac.uk/0596101996**

Web resources

I suggest you explore the following websites:

http://www.w3schools.com/js

http://www.javascriptkit.com

Looking at different JavaScript code examples is very important at this stage, and I encourage you to visit the previous websites as much in deep as possible, and to search for more examples in other websites.

This will allow you to better understand some of the capabilities of JavaScript.

At the same time, remember that you might find scripts that contain errors or statements that contrast with recommendations from the TT281 teaching material. Keep this in mind when answering your CMA questions and when working on your ECA.

It's quite challenging reading, but the **ECMAScript Specification** (the formal standard for much of JavaScript) is also available for download:

http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf