Dhananjay Kolate

JAVASCRIPT : BASIC SCRIPTING

JAVASCRIPT

Chapter 12. JavaScript 1: Basic Scripting

Table of Contents

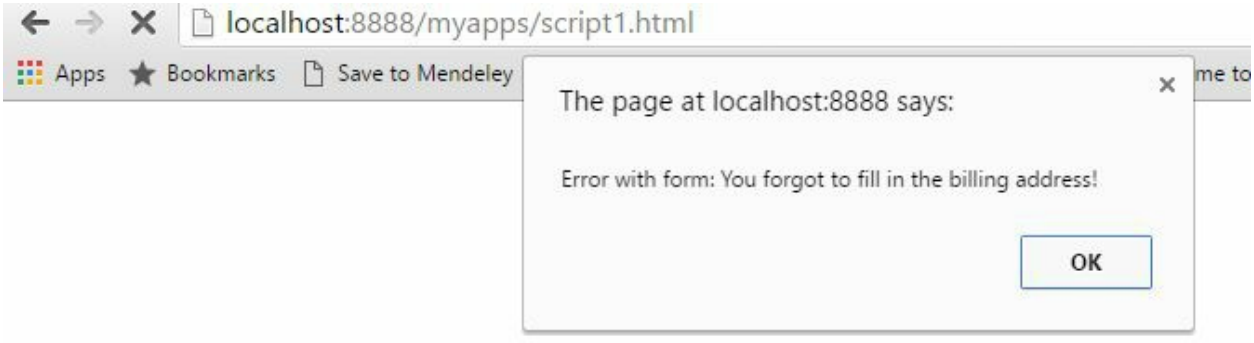
Objectives

At the end of this chapter you will be able to:
• Explain the differences between JavaScript and Java;
• Write HTML files using some basic JavaScript tags and objects.

11.1 Introduction

Web browsers were originally designed to interpret HTML with two primary purposes: to render documents marked up in HTML to an acceptable level of quality, and, crucially, to be able to follow hyperlinks to resources. As the Web grew, so did the demand for more sophisticated Web content. Among many other extensions, graphics, forms, and tables were added to the HTML standard. With the exception of forms, there is nothing in HTML that supports interaction with the user. Given the ubiquity of Web browsers, and the effort which millions of ordinary people have put into learning to use them, they provide an almost universal starting point for interacting with complex systems, particularly commercial, Internet based systems. Hence the need for sophisticated interaction facilities within Web browsers.

The main means for providing interactivity within HTML documents is the JavaScript programming language. HTML documents can include JavaScript programmes that are interpreted (i.e. run) by the Web browser displaying the Web document. In a real sense, JavaScript allows a Web document to interact with its environment — that is, with the browser that is displaying it. Ultimately, it lets the Web document become more interactive, to the user's benefit. For example, the following message could be given to a user when they submit a form with a missing field:



The above message can be shown with the following JavaScript code.

```
<SCRIPT>
window.alert('Error with form: You forgot to fill in the billing address!')
</SCRIPT>
```

The JavaScript code is contained within the <SCRIPT> and </SCRIPT> tags. Everything between those tags must conform to the JavaScript standard (the standard itself is an ECMA International standard, called ECMAScript). The above statement is an instruction to the browser requesting that an alert box display the message "Error with form: You forgot to fill in the billing address!".

This unit will later cover another way to include JavaScript in HTML documents. It is worth noting for now that the <SCRIPT> tag can include a language attribute to ensure the browser interprets the enclosed commands as JavaScript, since other languages have, in the past, been used (such as VBScript, which is no longer used in new websites, and is supported by very few browsers). For simplicity, we will use the attribute's default value (of JavaScript) by omitting the attribute from the <SCRIPT> tag.

11.1.1 Differences between JavaScript and Java

While programming is covered in the programming module of this course, JavaScript differs from Java in some important areas that we will quickly review. JavaScript objects are covered in more detail in later chapters, so we will not go into any great depth here.

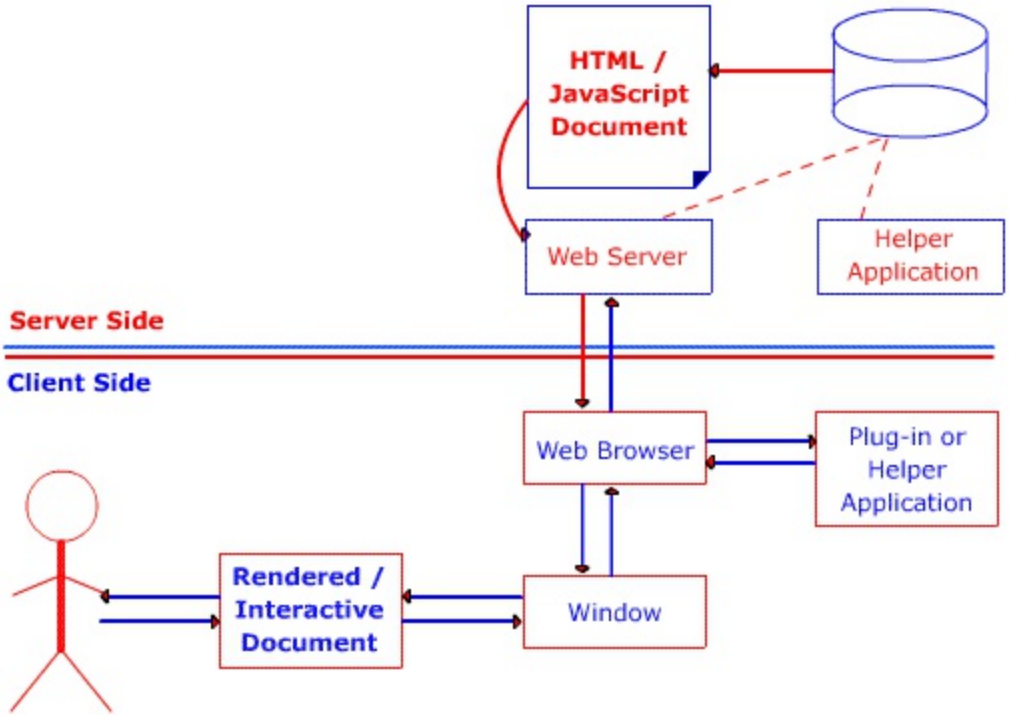
In Java, all functions must belong to a class, and for this reason are called methods. In JavaScript, a function does not have to belong to a particular object at all. When a function does, however, it is often called a method. Functions and methods are both implemented in the same way, using the function keyword. All methods are functions, but not all functions are methods.

Unlike Java, JavaScript does not contain the idea of classes. Instead, JavaScript has constructors, which are a special kind of function that directly creates objects. These constructor functions define the state variables which each object holds and initialises their values. These variables are often called the object's properties. Constructor functions also supply objects with their methods.

In JavaScript, functions are themselves a special kind of object called Function Objects. Function Objects can be called just as normal functions in other languages, but because they are objects they can themselves be stored in variables and can be easily passed around as, say, arguments to other functions. They can also contain properties of their own.Constructor functions have a special Prototype property which is used to implement inheritance, as will be explained later in the chapter on objects. Constructor functions are called using the new keyword when creating objects.

JavaScript communicates with its environment by calling methods on objects representing components of that environment, such as an object representing the window the HTML document is displayed in, an object representing the document itself, and so on. Ignoring the server side at present, we conceptually have a browser that interacts with a window, which interacts with a document, which is itself the user interface. JavaScript allows the user

interface to be programmed. This is accomplished by providing the means, in JavaScript, for a user to interact with a system via the document rendered in the browser window, as depicted below.



A user may request a document via an URL; a Web server delivers the document to a Web browser which not only displays it, but also executes any interactive elements.

Now do Review Questions 1, 2 and 3.

11.2 JavaScript within HTML

JavaScript statements are embedded within an HTML document and are interpreted by a Web browser. Unlike programming in Java, a programmer does not programme with JavaScript by preparing source code and compiling it to produce executable code. JavaScript is directly executed by the Web browser. The most general form of JavaScript used in HTML documents is to include JavaScript statements within <*SCRIPT*> and </*SCRIPT*> tags. Each JavaScript statement is written on a separate line. If a statement is in some way incorrect, the browser (and its builtin JavaScript interpreter) may report an error, or it may stop executing the erroneous JavaScript and do nothing. Let us examine a simple HTML4.0 document that does nothing that could not be done with HTML alone. All that it does is have the document include the italicised text "Welcome to programming with JavaScript!".

Most of the document is normal HTML. There are two new tags — <*SCRIPT*> to indicate the start of JavaScript code, and </*SCRIPT*> to indicate its end. The only line of JavaScript is the one containing:

document.write("Welcome to programming with JavaScript!")

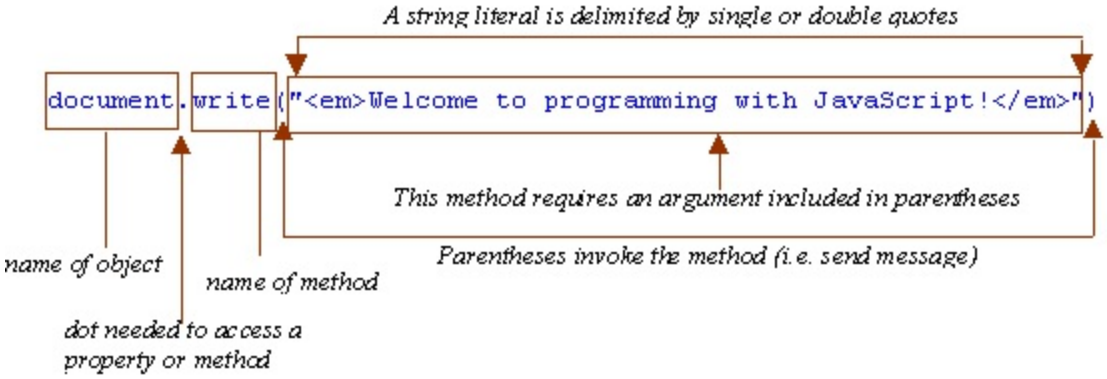
Activity 1

1. Using HTML5 tags create a file named script1.html with <HTML> and <BODY> tags. 2. Insert the following code and load the file to your browser.

```
<script>
document.write("<em>Welcome to programming with JavaScript!</em>")
</script>
```

What this mixture of HTML and JavaScript does is to make the browser switch from interpreting HTML to interpreting JavaScript when it encounters the </SCRIPT> tag. The line is interpreted as follows: first, it calls the write method of the document object with the argument "Welcome to programming with JavaScript!". The argument is a literal string containing valid HTML markup between the start and end emphasis tags (em> and) that cause what they delimit to appear in italics.

Let us examine the individual parts of the expression, as in the diagram below. Note that in this case the argument is a literal string.



As you might guess, all JavaScript does in this statement is to insert the argument text 'into' the document to be interpreted as HTML — hence the emphasis tags produce italics. Of course, the original document delivered by the server is not modified; nothing is actually written into the original document. What happens is that the JavaScript method inserts its argument in the stream of characters that the browser is interpreting.

Now carry out the following exercise and once you have studied its discussion continue with the unit's content.

Exercise 1

Modify the document in Activity 1 so that some HTML text is outputted immediately before and immediately after the welcome greeting. You can find a discussion of this exercise at the end of the unit.

11.2.1 Arguments

Arguments are also known as parameters. An argument is information that must be included with a function or method so that it may achieve its purpose. As in Java, some methods and functions are designed to have information supplied to them, so that their effect can differ in detail. If a function or method is designed in this way, it must be supplied with the required arguments when it is called. The document method write has been designed and implemented that way. Another function that needs a string argument is the window method alert, which was briefly shown earlier in these notes. The next exercise makes use of alert.

Exercise 2

Write the JavaScript statements that will show the welcome greeting in an alert box. (Hint: remember alert is a method of windowobjects, not of document objects.)
You can find a discussion of this exercise at the end of the unit.
document.write is an example of a function that can take an unspecified number of arguments. It must, however, have at least one. The following example uses more than one argument:

```
document.write("<BU>", "<LI>books</LI>", "<LI>magazines</LI>", "
</BU>")
```

In JavaScript, as in Java, multiple arguments to a function must be separated by commas and provided in an order determined by the function. There is no simple rule to tell how many arguments a method must have or what their order should be, and this information can only be obtained from the method's definition, or from a good reference document or book.

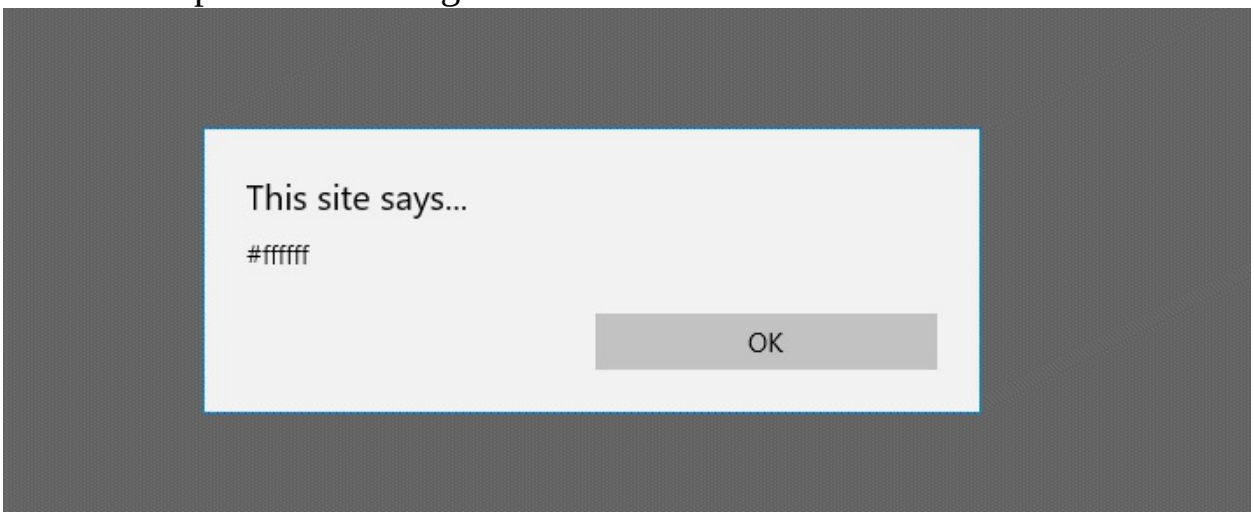
To Do

Find a JavaScript Reference source that has information about the objects, properties and functions available to a JavaScript programmer.
Now do Review Questions 4, 5 and 6.

11.2.2 Accessing and Changing Property Values

In JavaScript, it is commonplace to directly access or change the value of a property. One such document property is *bgColor*, which represents the document's background colour. Directly accessing a property is similar to accessing a method: write the object name followed by a period (a dot) and the property name. Thus, to access a document's background colour, write *document.bgColor*. Note the American English spelling of 'Color', and the use of a capital 'C' inside the word. The spelling and capitalisation must be precise or a browser's JavaScript interpreter will not properly process the script.

Of course, methods can be combined with the ability to access properties. For example, the *alert* method of *window* can be used to report the document's background colour:
window.alert(document.bgColor)
This would produce a dialogue box as follows.



The box shows the hexadecimal (HEX) representation of the colour, with *#FFFFFF* being the HEX for white, a document's default background colour. The content of the dialogue box can be made more comprehensible by announcing the value with another string explaining the value. One way to do this is with the string concatenation operator, +, which combines what precedes it with what follows it. So, for example, *"changing " + "colour"* results in the string *"changing colour"*. Note the space at the end of the first

string literal appears between the two words in the string resulting from using +. Hence you can concatenate *"background colour is (in Hex): "* with *document.bgColor* to produce more helpful output:

window.alert("background colour is (in Hex): " + document.bgColor)

To change a document's background colour, simply assign the new value to the property using the = operator. Do not confuse this symbol with equality. JavaScript belongs to a class of programming languages (just as Java) that use = as the assignment operator: it assigns the value of whatever is to its right to whatever is on its left. Another symbol is used for equality testing, as we discuss later. Hence, to set the background colour of a document to blue (HEX 0000FF), we can use:

document.bgColor = "0000FF"
JavaScript provides a set of mnemonic strings for many colours. For example: document.bgColor = "blue"

Note that no matter what value was used to set a colour, if you access it as above the output will always be in HEX. (This is because JavaScript automatically converts between different types of data. We will see this a lot in JavaScript programming.)

Activity 2: Checking and Setting Background Colour

Write and test a document containing HTML and JavaScript that displays the background colour of the document, warns you of a change of colour is to take place, then changes it to blue (use the string *"blue"*). Repeat this for the orange-like colour coral, using its HEX representation *"FF7F50"* and confirm the change with an alert.

You can find a discussion of this activity at the end of the unit.

Activity 3: Setting a Document's foregrond colour

JavaScript considers the default colour of a document (i.e. the colour of its text) to be the property *fgColor*. Write a script to set the background colour of

a document to blue and the foreground colour to white. You can find a discussion of this activity at the end of the unit.

11.2.3 Variables

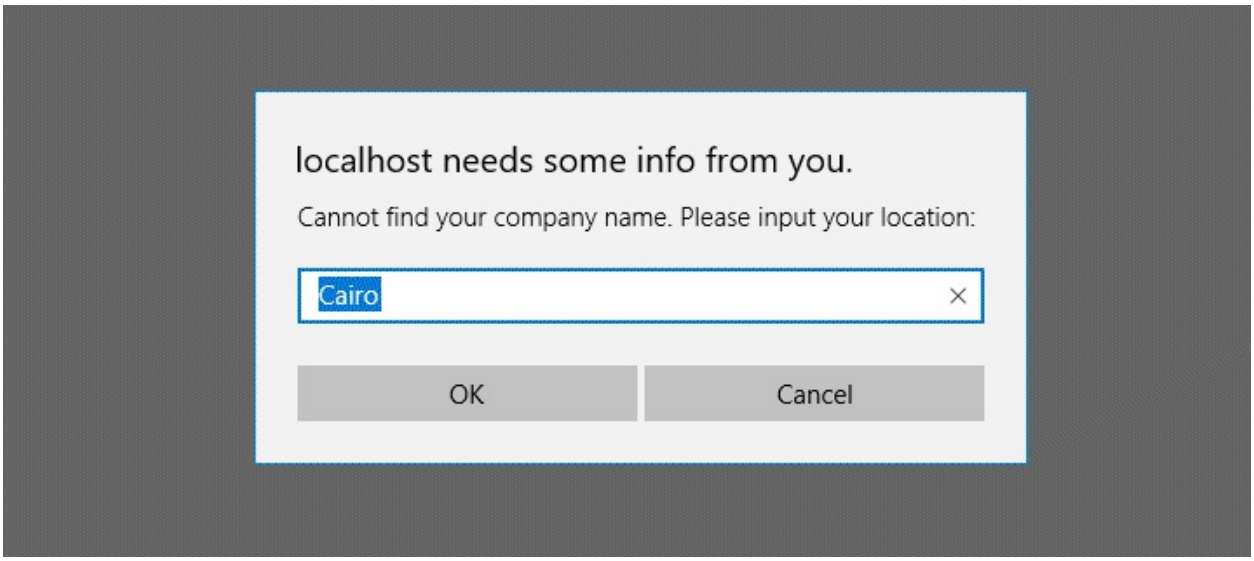
With the exception of alert dialogue boxes, so far we have used JavaScript to do nothing more than what HTML can do. We now proceed to the vital concept of programming with variables via another interactive facility — dialogue boxes that allow the user to enter data.

Exercise 3

An important principle of object-oriented programming is that any facilities provided by an object should be encapsulated within the object. Remembering which object provides the *alert* method, which object do you think has a method for prompting the user for input in a dialogue box. You can find a discussion of this exercise at the end of the unit. The *prompt* method allows you to make a window display a dialogue box containing an explanation prompt and a default value for input.

For example, the statement below explains to the user that their company name cannot be found (let's assume some previous processing has determined this). It provides a suggested default of Cairo, which we assume makes sense for the Web application.

window.prompt("Cannot find your company name. Please input your location:", "Cairo")
Note that the two string arguments, each delimited by their own quotes, are separated using a comma. The output from a document containing this JavaScript appears as run on Microsoft Edge.



So, what might be done with input obtained this way? In general, we would use it to change the state of some part of the Web application, and such a simple change of state could eventually lead to more complex behaviour. One straightforward use is to modify the HTML stream to include the input obtained by *window.prompt*. For example, imagine an application that generated an email message whose content is an HTML document. We could ask for the subject and then write it as a level 1 heading (i.e. using *<H1>* and *</H1>*):

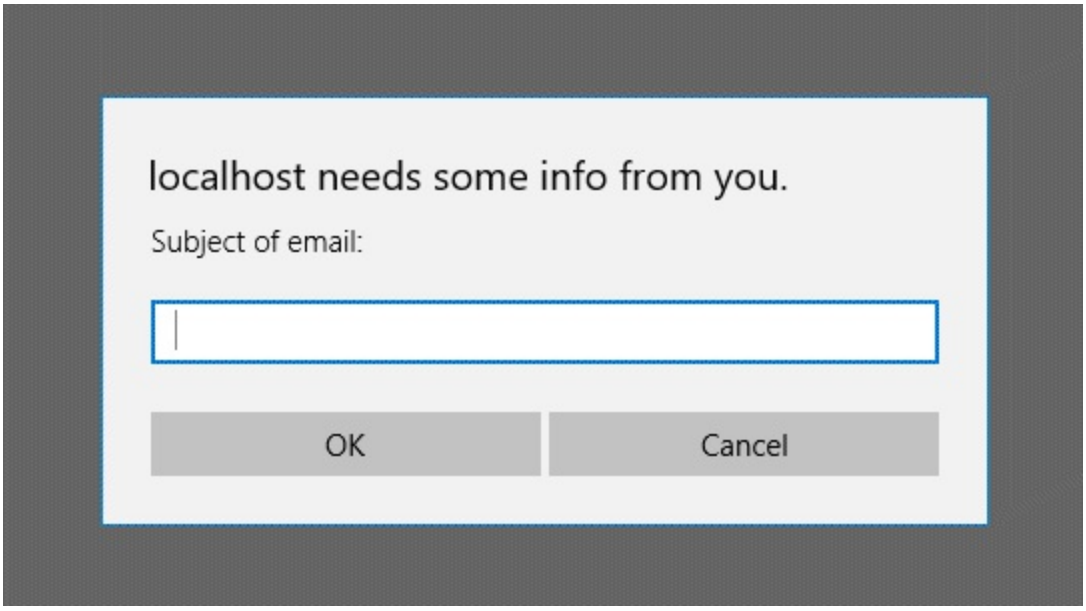
```
document.write("<H1>Subject: "+window.prompt("Subject of email:","")+"
</H1>")
```

This rather complicated argument to document.write needs explaining. The complication comes from needing to evaluate the prompt to determine the argument to write as a whole. When this statement is executed, the interpreter tries to concatenate the three strings:

1. "<H1>Subject: "
2. window.prompt("Subject of email:","") 3. "</H1>"

String 1 is the first part of the heading; it includes the start tag for the level 1 heading, and the beginning of the heading — 'Subject: '. String 3 is the end tag for the level 1 heading. String 2 cannot be part of the concatenation for the *write* method unless it is itself evaluated; hence the following dialogue box is produced. Notice how no default for the input is provided, because the

second argument is an empty string — a string with zero characters in it.



If the user types in 'Unable to book flight' and clicks OK (or presses the Enter key), then the *prompt* method delivers that string object as String 2 and the concatenation can be completed, resulting in the heading in the browser below:

An alternative way to programme this type of behaviour is to break it up into smaller pieces and to provide the means to remember the result of evaluating some complex expression, then using that result later in the programme in whatever way is needed. This is what variables are for: they are names that can be used to refer to an object or value for a variety of purposes. They are called variables because they can be used to refer to different objects at different times of the programme's execution. As it happens, in JavaScript a variable can refer to objects of any type, whether they be strings, numbers, or user defined objects. This is very flexible and powerful, but it means that the interpreter cannot help you by restricting the functionality of the variable to a

declared purpose, as in some other languages (such as Java). The most general way to use a variable is to declare it with a special key word, *var*, as in:

var input

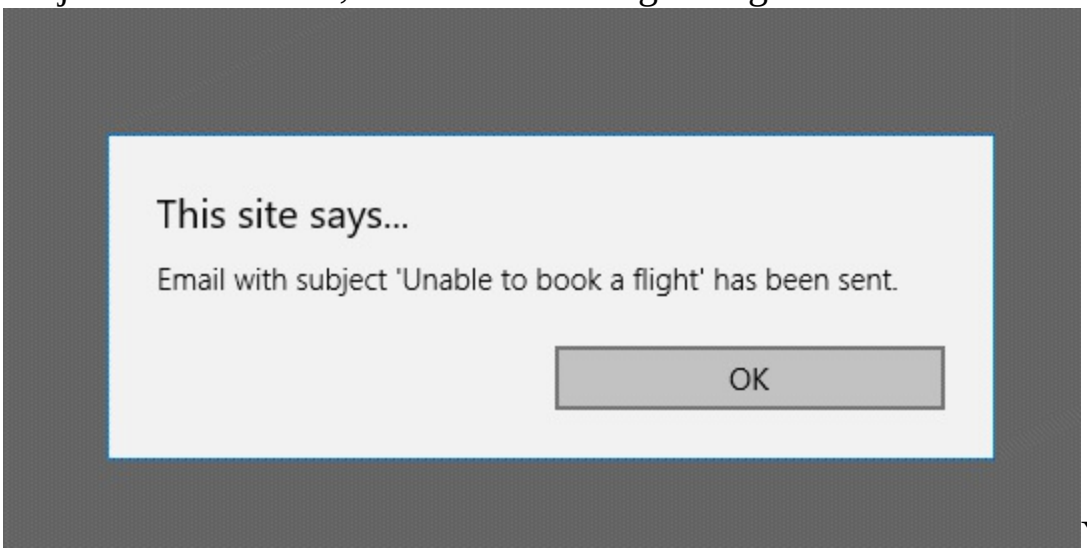
Once a variable has been declared it can be made to refer to an object with the assignment ope r a t or , =, introduced earlier. Now the email subject prompter can be reprogrammed as follows (below, we choose the obvious variable name — *input* — which has nothing to do with the HTML tag or the JavaScript object):

```
<SCRIPT>
var input
input = window.prompt("Subject of email:","") document.write("
<H1>Subject: " + input + "</H1>")
</SCRIPT>
```

Executing this script has the same effect as the earlier version that included the prompt as an argument to *write*. A user cannot tell the difference.

Exercise 4

Write another script to be included in the same document as the previous one so that the new script produces a confirmation that the email with the given subject has been sent, as in the following dialogue box.



You can

find a discussion of this exercise at the end of the unit.

11.2.4 JavaScript Comments

In the same way that HTML provides the means to prevent an HTML interpreter (a browser) from acting on text in a document, JavaScript also provides the means to turn its statements into comments that the JavaScript interpreter ignores. Because of its C++, C and Java heritage, JavaScript accepts comment syntax from all of these languages: anything between /* and */ is ignored, as is everything on a line following //. The former brackets a comment, which can be over several lines or within a statement. The latter is typically used when the end of a line contains a comment. For example, the above script can be commented as follows:

<SCRIPT>
var input // this variable will be used in a later script input =
window.prompt("Subject of email:","")
/* We have to generate HTML according to company style guidelines found in
Document 1998-EM-GD-V3.1, hence what follows. */ document.write("
<H1>Subject: " + input + "</H1>") </SCRIPT>

JavaScript is used within HTML and it is possible that older browsers are unable to run JavaScript, or that a user has disabled the JavaScript functionality of a recent browser. Therefore JavaScript also understands the opening HTML comment, but not the closing comment. This allows JavaScript code to use a mixture of HTML comments to allow the JavaScript to be ignored by browsers that cannot run JavaScript and would format it as HTML text.

The previous example could be changed so that it works in a JavaScript-enabled Web browser but has no effect on the output of a Web browser that cannot handle JavaScript. The line immediately after the <SCRIPT> tag is the start of an HTML comment, which is ignored by both language interpreters. The line before the </SCRIPT> tag is ignored by a JavaScript interpreter, but if there was none, the HTML interpreter would have been ignoring everything up to the closing HTML comment bracket.

```
<SCRIPT>
<!---the JavaScript can be ignored
var input // this variable will be used in a later script input =
window.prompt("Subject of email:","")
/* We have to generate HTML according to company style guidelines found
in
Document 1998-EM-GD-V3.1, hence what follows. */
document.write("<H1>Subject: " + input + "</H1>")
//--->
</SCRIPT>
```

Activity 4: using user input to set colours

Write and test an HTML/JavaScript document to prompt the user for background colour and foreground colour and then output sample text. You should use appropriately named variables and maybe reuse them when generating the sample text. When testing, make sure you have a variety of inputs — valid and invalid colours, and cancelled inputs. Note down what happens in each case.

You can find a discussion of this activity at the end of the unit.

Activity 5: dealing with errors

The following is bad JavaScript. Type (or copy) it into a suitable HTML document to explore what goes wrong when you try to interpret this in a browser. Correct each problem statement as you see necessary and write down what you did and why. (Hint: look at the use of the *write* method from which you can infer what the script is to do.) Also identify anything that seems unnecessary and anything you think is wrong at first sight but which might turn out to be correct. Note that different browsers (different JavaScript interpreters) will report different problems. If you have access to more than one browser you may like to see which provides most help.

Note that you are not expected to understand all the problems in the script below. What you are expected to understand is that you can 'instrument' the code with suitable *write* methods or *alert* methods (like the ones you have

already used) to help you reason about what is happening (or not happening). So be prepared to temporarily add statements to the code so you can trace how it is being interpreted. And be prepared to spend quite a bit of time exploring the code.

```
var fore, back
back = document.bgcolor
fore = document.fgColor = fore
newFore = window.prompt("Foreground colour:",fore) document.fgColor =
newFore
input = window.prompt("What do you want written out underlined in colour
document.write("You typed: " + <U> + "input + </U>")
```

You can find a discussion of this activity at the end of the unit.

Activity 6: The confirm method

As well as the *alert* and *prompt* methods, the *window* object provides a *confirm* method that produces a dialogue box with the buttons OK and Cancel. Like alert, it takes a string argument that is displayed in the dialogue box. Explore this method by writing HTML/JavaScript documents that use it.

You can find a discussion of this activity at the end of the unit. Now do Review Questions 7, 8 and 9.

11.3 Some Basic JavaScript Objects

So far we have used the *Window* and *Document* objects. We now list many of the most frequently needed properties of *Window* and *Document* and all of the details of *Date*, which we have not yet used.

You do not have to memorise these lists. You will eventually remember what you most often use. However, it is important to spend time reading reference material because sometimes you will need some facility and will need to be aware if it exists or not. Later activities assume you can use most of the facilities described below.

Notice that some properties are described as an 'array' with square brackets. An array is a collection of items numbered from zero, as in Java, and is accessed in the same way, using a suffix containing the index number in square brackets. For instance, *window.frames[2]* refers to the third frame in a window. Arrays are covered in more detail in a later chapter.

First, note that variables *window* and *document* are automatically declared so that we have been able to make use of them when scripting. They refer to the current window and its document respectively. Automatic declaration, however, is not usual. For example, to use a *Date* object you will have to create one with a constructor, as described later.

Note that where a method requires an argument or arguments, these arguments are indicated by a phrase in angle brackets that list the arguments that are needed. For example, <*string*> would indicate that a string argument is needed. (Do not confuse these with HTML tags.)

11.3.1 Window Objects

The following table lists frequently used properties and methods that are supported by Firefox, Internet Explorer and Opera.
Window Description
Properties (Partial List) **defaultStatus** — a string that specifies what appears in the window's status area
document — a reference to the document object contained by the window
frames[] — an array of frames in the window's document
length — the number of elements in the frames array, i.e. the number of frames
location — a reference to the Location object for the window
Math — a reference to a mathematical object that provides various mathematical functions
name — a string containing the name of the window
Window Description
self — a reference the window itself **status** — a reference to the window's status area
top — a reference to the top-level window containing this one, only if this one is a frame

Methods (Partial List) **alert(<string>)** — produced a dialogue box containing the string and a single button labelled OK

close() — close the window
confirm(<string>) — ask a yes/no question with the string argument

moveBy(<number in x>, <number in y>) — move the window by the given number of pixels in the x (horizontal) and y (vertical) directions

moveTo(<number for x>, <number for y>) — move the window to the location given for x (horizontal) and y (vertical) directions

prompt(<string for prompt>, <string for default>) — prompt with OK and Cancel buttons using the first string as prompt and the second as default for input

resizeBy(<number in x>, <number in y>) — resize the window by the given number of pixels in the x (horizontal) and y (vertical) directions

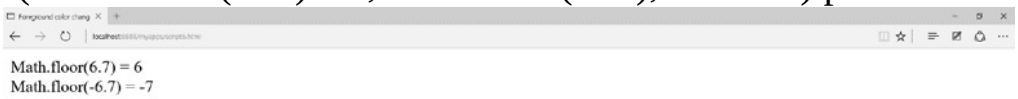
resizeTo(<number for x>, <number for y>) — resize the window to the size given for x (horizontal) and y (vertical) directions

scrollTo(<number for x>, <number for y>) — scroll the document in the window so that the position given for the x (horizontal) and y (vertical) direction is in the tope left corner of the window

Note that because the *Window* object (referred to by the *window* variable) is the main object of the system as far as JavaScript is concerned, its name can be left out. Therefore we could have simply written *confirm("Click on OK or Cancel or close box")*. However, we recommend you explicitly use the *window* object.

There are some exceptions to this recommendation. The first is the use of the *Math* object ('math' being the American abbreviation for 'mathematics'). Because the window contains a *Math* object as a property, you can refer to it simply using *Math*. For example *Math.floor(<number>)* allows you to truncate (i.e. round down) a real number to the next smaller integer (not the closest to zero). Hence:

document.write("Math.floor(6.7) = ", Math.floor(6.7), "
")
document.write("Math.floor(-6.7) = ", Math.floor(-6.7), "
") produces



the following:

11.3.2 Document Object

The list of *Document* properties given below is almost complete (excluding those specific to particular browsers). Many are arrays, or are associated with other objects not yet encountered. Do not expect to use these in this unit.
Document Description
Properties (Partial List) **aLinkColor** — string specifying colour of active links
anchors[] — an array of Anchor objects **applets[]** — an array of Java (not JavaScript) applets (one for each HTML <APPLET> tag)
bgColor — background colour of document
cookie — a string associated with document cookies
domain — specifies the Internet domain that was the source of the document
embeds[] — an array of embedded objects (one for each HTML <EMBED> tag)
fgColor — foreground colour of document text
forms[] — an array of Form objects (one for each HTML <FORM> tag)
images[] — an array of Image objects (one for each HTML <IMG> tag)

lastModified — a string specifying the date of the last change to the document as reported by its Web server

linkColor — string specifying colour of unvisited links
Document
Method (Partial List)
Description
links[] — an array of Link objects (one for each hypertext link)
location — a synonym for the URL property (not the same as window.location)
plugins[] — a synonym for the embeds[] property

referrer — a string specifying the URL of the document containing the link to this one that the user followed to reach the document

title — title of the document (may not be changed)
URL — a string containing the URL of the document (may not be changed)

vlinkColor — string specifying colour of visited links
close() — closes the document stream opened using the *open()* method

open() — open a stream to which document contents can be written, i.e. to which output of subsequent *write()* or *writeln()* methods will be appended

write(<at least one argument>) — appends to current output stream; takes any argument that can be converted to a string

writeln(<at least one argument>) — same as *write()* but adds a new line to output that usually has no effect for HTML, except if after <PRE> tag.

11.3.3 Date Objects

Date objects have properties and methods for handling date and time. To access dates or times a Date object must be created, typically using *new Date()*. Facilities are provided for both the local date and time and universal date and time (UTC) or Greenwich Mean Time (GMT). For example, the following JavaScript code creates a new *Date* object as part of the initialization of the variable *today*.

```
var today = new Date()
document.write('1 -- ', today, '<BR>')
document.write('2 -- ', today.toGMTString(), '<BR>') document.write('3 -- ',
today.toLocaleString(), '<BR>')
```

The following is the output that results in Microsoft Edge.

The following table lists the attributes and methods of *Date*. Note that Date has no properties. Usually you can interchange the terms attribute and property. In English they mean more or less the same thing. However, in object technology an attribute is simply an idea you name because you expect it is somehow represented by the object. Often an attribute is represented in JavaScript as a property, but not in this case. You cannot directly access attributes of a date like the hour or minute; you must use accessor methods, such as the methods that begin with 'get' or 'set' below.

Date

Date Constructors *Date* objects have attributes and methods for handling date and time. To access dates or times a Date object must be created, typically

using *new Date()*. Facilities are provided for both the local date and time and universal date and time (UTC) or Greenwich Mean Time (GMT).
There are four forms of constructor for *Date*

Date Properties 1. **new Date()** In the first form, a Date object is created initialised to the current date and time.

2. **new Date(time in milliseconds)** In the second form, the date and time given to the object created is specified by the time in milliseconds from 1 January 1970, GMT. This is usually used when the number of milliseconds has been computed using various Date methods.

3. **new Date(string in date format)** In the third form an appropriately formatted string is used
to set the date and time of the newly created object.

4. **new Date(year, month, day, hour, minute, second, millisecond)** In the fourth form between two and seven numbers are given to specify the date and time, attribute by attribute (see under Properties).

There are no properties that can be get or set directly. All attributes of a *Date* object must be accessed via its methods. The *Date* attributes are:

year — a four digit number
month — an integer between 0, for January, and 11, for December
day — an integer between 1 and 31 specifying the day of the month
hour — an integer between 0, for midnight, and 23, for 11pm
minute — an integer between 0 and 59 that specifies the minute in the hour

second — an integer between 0 and 59 that specifies the second in the hour
millisecond — an integer between 0 and 999 that specified the millisecond

Date Methods **getDate()** — returns the day attribute, i.e. theday of the month between 1 and 31
getDay() — returns the day of the week, 0 for Sunday, 1 for Monday, ..., 6 for Saturday
getFullYear() — returns the year attribute (getYear() is deprecated version)
getHours() — returns the hour attribute (note plural in method name)

getMilliseconds() — returns the millisecond attribute (note plural in method name)
getMinutes() — returns the minute attribute (note plural in method name)
getMonth() — returns the month attribute
getSeconds() — returns the second attribute (note plural in method name)
getTime() — returns the date as the number of milliseconds since midnight, 1 January 1970

getTimezoneOffset() — returns the time zone difference in minutes between date and GMT **getUTCDate()** — returns in universal time the day attribute, i.e. the day of the month between 1 and 31

getUTCDay() — returns in universal time the day of the week, 0 for Sunday, 1 for Monday, ..., 6 for Saturday

getUTCFullYear() — returns in universal time the year attribute
getUTCHours() — returns in universal time the hour attribute (note plural in method name)
getUTCMilliseconds() — returns in universal time the millisecond attribute (note plural in method name)
getUTCMinutes() — returns in universal time the minute attribute (note plural in method name)
getUTCMonth() — returns in universal time the month attribute
getUTCSeconds() — returns in universal time the second attribute (note plural in method name)
setDate(<number 1-31>) — sets the day attribute, i.e. the day of the month between 1 and 31
setDay(<number 0-6>) — sets the day of the week, 0 for Sunday, 1 for Monday, ..., 6 for Saturday
setFullYear(<four-digit number>) — sets the year attribute (setYear() is deprecated version)
setHours(<number 0-23>) — sets the hour attribute
(note plural in method name)
setMilliseconds(<number 0-999) — sets the millisecond attribute (note plural in method name)
setMinutes(<number 0-59>) — sets the minute attribute (note plural in method name)

setMonth(<number 0-11>) — sets the month attribute
setSeconds(<number 0-59>) — sets the second attribute (note plural in method name)

setTime(<non-negative number>) — sets the date as the number of milliseconds since midnight, 1 January 1970

setUTCDate(<number 1-31>) — sets in universal time the day attribute, i.e. the day of the month between 1 and 31

SetUTCDay(<number 0-6>) — sets in universal time the day of the week, 0 for Sunday, 1 for Monday, ..., 6 for Saturday

setUTCFullYear(<four-digit number>) — sets in universal time the year attribute
setUTCHours(<number 0-23>) — sets in universal time the hour attribute (note plural in method name)

setUTCMilliseconds(<number 0-999) — sets in universal time the millisecond attribute (note plural in method name)

setUTCMinutes(<number 0-59>) — sets in universal time the minute attribute (note plural in method name)
setUTCMonth(<number 0-11>) — sets in universal time the month attribute
setUTCSeconds(<number 0-59>) — sets in universal time the second attribute (note plural in method name)

toGMTString() — returns a string representation of the date using GMT, e.g. in Internet Explorer 5: "Sun, 5 Sep 1999 17:21:48 UTC"

toLocaleString() — returns a string representation of the date using local time zone,
e.g. in Internet Explorer 5: "09/05/1999 18:21:48"

toString() — returns a string representation of the date, e.g. in Internet Explorer 5: "Sun Sep 5 18:21:48 UTC+0100 1999"

toUTCString() — returns a string representation of the date using universal time, e.g. in Internet Explorer 5: "Sun, 5 Sep 1999 17:21:48 UTC"

value Of() — returns the date as the number of milliseconds since midnight, 1 January 1970

Activity 7: Changing the window status

When you move the mouse pointer over a hyperlink, the status area in the bottom left of the browser window normally changes to reveal the link's URL. User often use this information to decide whether to follow the link. You could, instead, arrange for the status area to display some marketing information by setting the *status* property of the *window* object.

Write a HTML/JavaScript document that includes an ordinary anchor containing an unwelcome URL and a JavaScript script that assigns a more enticing string to the status portion (e.g. Sale: all USB devices at 50% discount).

You can find a discussion of this activity at the end of the unit.

Activity 8: Semicolons to end statements

We have been using the practice of ending JavaScript statements with line breaks. Strictly speaking, a statement is terminated by a semicolon (;). When followed by a new line, the semicolon can be omitted. The following statements take advantage of this rule.

document.bgColor = "blue" document.fgColor = "white"
document.write("This text should be white on a blue background.")
Rewrite the above statements to use semicolons. You can find a discussion of this activity at the end of the unit.

Activity 9: including separate JavaScript files

The <SCRIPT> tag may have an optional SRC attribute to specify the URL of a text file containing JavaScript statements that are to follow the tag.

Rewrite the sample solution given for Activity 2 using this facility. (Hint: use a file in the same directory as the HTML file.)

Note that by convention an extension of .js is used as a suffix to the name of the JavaScript file, e.g. *foreColour.js*.
You can find a discussion of this activity at the end of the unit.

Activity 9: Opening a new Window

Check what happens when you open a new window with the following JavaScript. Try to predict what will happen before trying this. In particular, try to predict what object the variable *document* refer to.

window.open() document.bgColor="blue" document.fgColor = "yellow" document.write('This is the new document')

You can find a discussion of this activity at the end of the unit. Now do Review Questions 10 and 11.

11.4 Review Questions

What is the main advantage of JavaScript?

11.4.1 Review Question 1

What is the main advantage of JavaScript?
You can find the answer to this question at the end of the unit.

11.4.2 Review Question 2

What is the JavaScript term for the parts of an object's state, and what is the term for a function that is a part of an object?
You can find the answer to this question at the end of the unit.

11.4.3 Review Question 3

Write down in your own words what happens when an interactive HTML/JavaScript document is loaded by a browser from a Web server. You can find the answer to this question at the end of the unit.

11.4.4 Review Question 4

What object does the method *alert* belong to? What does the method do in that object? You can find the answer to this question at the end of the unit.

11.4.5 Review Question 5

Explain what an argument is and give an example of one. You can find the answer to this question at the end of the unit.

11.4.6 Review Question 6

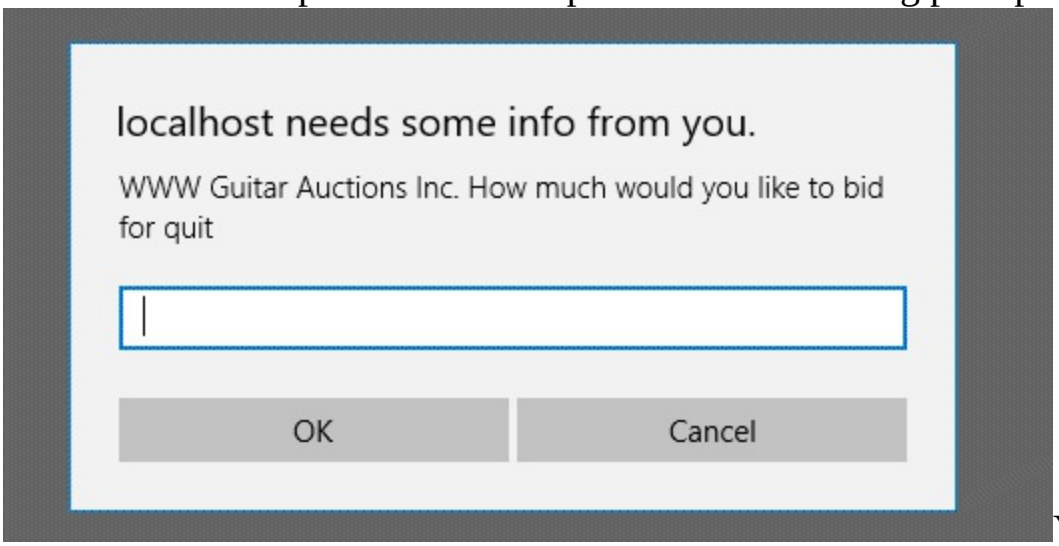
Write a statement in JavaScript that displays a dialogue box with the greeting 'Form ready for your application. Please proceed.'
You can find the answer to this question at the end of the unit.

11.4.7 Review Question 7

What is the purpose of a variable? How do you declare a variable in one script in a document for it to be used by another?
You can find the answer to this question at the end of the unit.

11.4.8 Review Question 8

Write the JavaScript code that will produce the following prompt:



You can find the answer to this question at the end of the unit.

11.4.9 Review Question 9

The date Fri Jan 4 11:38:00 UTC 1991 is 662989080980 milliseconds after midnight, 1 January 1970. How would you create a *Date* object referred to by variable *theDate*, which represented the date Fri Jan 4 11:38:00 1991 in universal time. You can find the answer to this question at the end of the unit.

11.4.10 Review Question 10

To which object does the method close() belong in the following line of JavaScript?
close()
You can find the answer to this question at the end of the unit.

11.4.11 Review Question 11

The window object includes a method resize that takes two arguments which specify the width and height of the window in pixels. Read the following script and write down what it does. (Do not try to work out what happens in 'odd' situations, like when a negative number is given.)

```
var squareSize
= window.prompt("In what size square would you like this?", "300")
window.resizeTo(squareSize,squareSize)
```

You can find the answer to this question at the end of the unit.

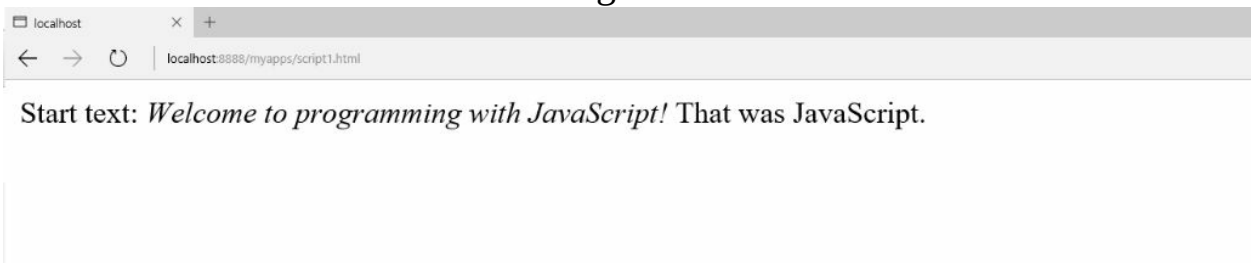
11.5 Discussions and Answers

11.5.1 Discussion of Exercise 1

It really does not matter what you display just before and just after the welcome greeting. The following HTML and JavaScript is the same as the original but with *Start text*: before and *That was JavaScript.* after. Without the document's preamble and closing tags we have:

Start text:
<SCRIPT>
document.write("Welcome to programming with JavaScript!")
</SCRIPT>
That was JavaScript.

This result when run in Microsoft Edge is:



Can you explain the spacing? That is, can you see why there is no space between the ':' and the 'W', and why there is no space between the '!' and the 'T'?

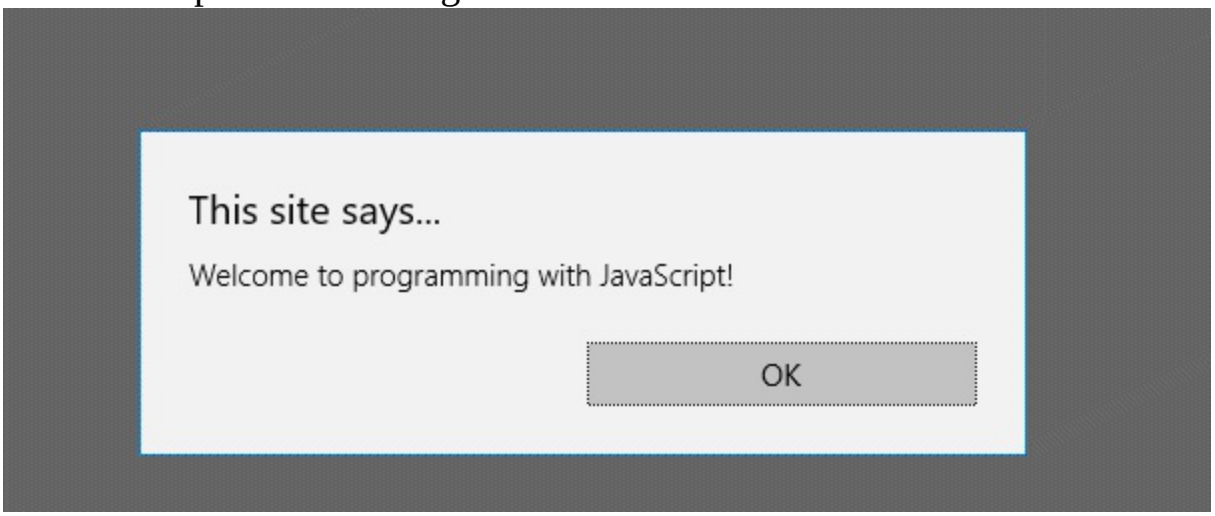
The answer is that the <SCRIPT> and </SCRIPT> tags do not themselves affect the HTML stream. If they, and what they enclose, had not been there, there would be no space between the ':' and the 'T'. If you were to put the tags on separate lines, then space would have been introduced by the line breaks. Of course, you could have included spaces or other spacing tags in the argument to write — like
.

11.5.2 Discussion of Exercise 2

Instead of using the write method with the greeting as parameter, use the alert method with the greeting as parameter, thus:

<SCRIPT>
window.alert("Welcome to programming with JavaScript!") </SCRIPT>

This would produce a dialogue box similar to the one below.



You can see that the <*em*> and </*em*> tags have been omitted. The *alert* method (of the object *window*) does not get the browser to interpret its argument as HTML, so the tags would appear as ordinary text in the dialogue box if we had included them.

Note that different versions of JavaScript implement their facilities a little different from each other. This was run in Microsoft Edge.

11.5.3 Discussion of Exercise 3

Given that the *window* object is the one that provides for the alert facility, it is reasonable to assume that it will supply the similar facility of prompting the user. Indeed, this is the case: the method *prompt* is provided. It requires two arguments — a string to prompt the user with an explanation inside the dialogue box, and a second one to act as default input.

11.5.4 Discussion of Exercise 4

Since the variable *input* still refers to what data the user gave in response to the prompt on email, it can be used in a subsequent script in the same document, thus:

```
<SCRIPT>
window.alert("Email with subject '" + input + "' has been sent.")
</SCRIPT>
```

There are several points to note about this script. First, is that there is no need to redeclare the variable *input*; redeclaring it would have had no effect. Second is the use of single quote marks within the double quotes that are used here to delimit strings. In fact, either can be used as delimiters, allowing you to use whatever is not a delimiter inside the string. Third as far as the JavaScript interpreter is concerned, the spaces either side of the concatenation operator (+) are irrelevant; they are there to help the human reader and are ignored by the interpreter.

11.5.5 Activity 2: Checking and Setting Background Colour

```
<!DOCTYPE html>
<!-- Copyright (c) 2015 UCT -->
<html>
<HEAD>
<TITLE>
Welcome message
</TITLE>
</head>
<BODY>
<SCRIPT>
window.alert("background colour is (in Hex): " + document.bgColor)
window.alert("changing colour to blue")
document.bgColor = "blue"
window.alert("background colour is (in Hex): " + document.bgColor)
window.alert("changing colour to coral") document.bgColor = "FF7F50"
window.alert("background colour is (in Hex): " + document.bgColor)
</SCRIPT>
</BODY>
</html>
```

11.5.6 Activity 3: Setting a document's foreground colour

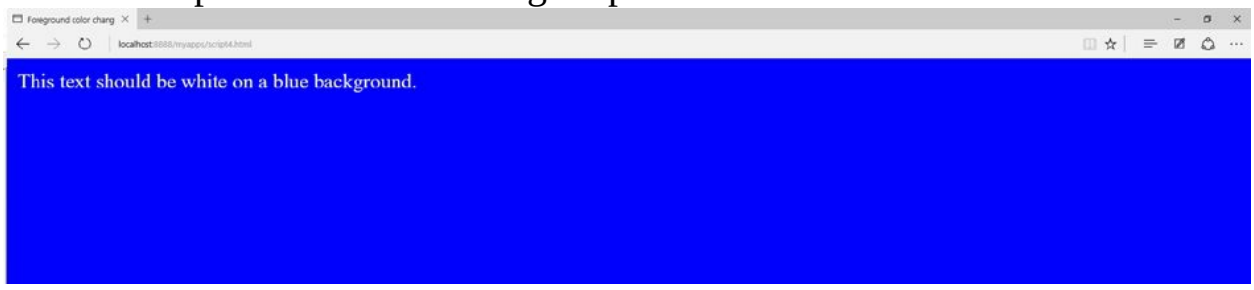
The following will change background colour and foreground colour as required and write some test text.

```
<!DOCTYPE html>
<!-- Copyright (c) 2015 UCT -->
<html>
<HEAD>
<TITLE>
Foreground color changes
</TITLE>
</head>

<BODY>
<SCRIPT>
document.bgColor = "blue"
document.fgColor = "white"
document.write("This text should be white on a blue background.")

</SCRIPT>
</BODY>
</html>
```

This would produce the following output.



11.5.7 Activity 4: Using user input to set colours

The following JavaScript does what is required. In this version, default colours are included in the prompt. These are not necessary.

```
var fore, back
back = window.prompt("Background colour:","coral") document.bgColor =
back
fore = window.prompt("Foreground colour:","blue") document.fgColor =
fore
document.write("This text should be "+ fore + " on a " + back + "
background")
```

Note the use of the variables *fore* and *back* for the foreground colour and background colour respectively. These are declared in a single *var* statement (you could have used two). Each is used to remember the data input via the *prompt* method and is subsequently used for two purposes: first, to set the associated property, and second, to form the sample text that allows you to confirm you have programmed the correct behaviour.

Supplying valid colours (e.g. green and yellow) other than any defaults you supply are sensible test cases. However, you must check what happens with not-so-sensible cases. For example, if you input both colours to be the same, you will not see the sample text. (However, if you use Select All from an appropriate browser menu, you should see the sample text.) If you input nonsense for either colour (e.g. %^RTkz) the variables will be assigned the nonsense value, but the properties will be changed to HEX 000000 which is the colour black.

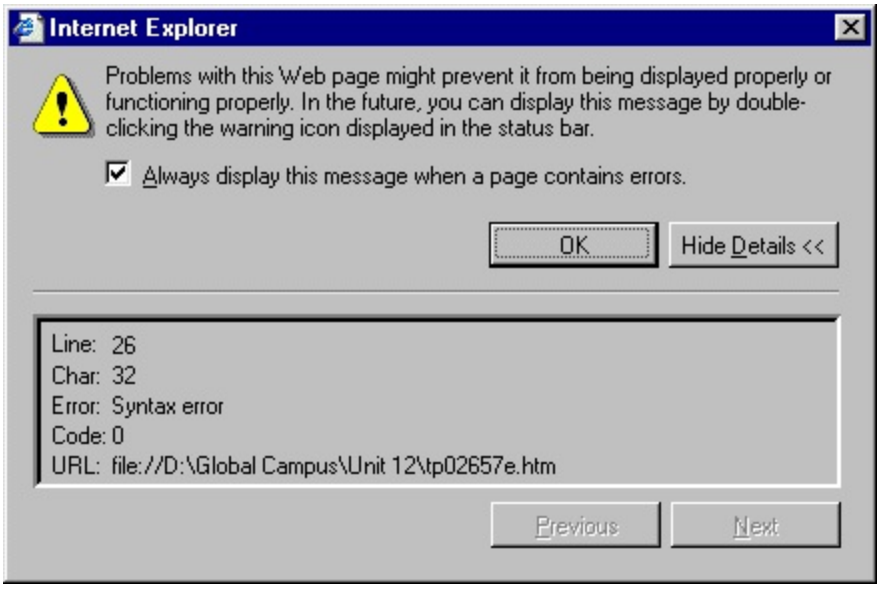
Something quite different happens if you click Cancel on the prompting dialogue boxes. When Cancel is clicked the prompt method returns the special value null — essentially meaning 'no value'. If null is assigned to either of the colour properties, they are left unchanged (although the variables still have the value *null*). This behaviour is not easy to determine.

There is a general lesson to be learnt here: the variables you use may be set to

something that does not make sense for their subsequent use and the values assigned to properties may not be the values they are given. Test thoroughly! And keep your test cases and notes of results in case you have to change your system.

11.5.8 Activity 5: Dealing with errors

The first thing to look at is what the script is supposed to do. The last statement in which a string is to be written into an HTML stream would suggest that some input (represented by the variable *input*) is to output underlined (you can recognise the <U> and </U> tags). In fact that is where the first problem is. Internet Explorer produces the following. In fact it does this before it executes the beginning of the script. (From this you can infer that the Microsoft interpreter does some checking of the entire script before interpreting any of it.)



In fact Netscape Navigator does not complain at all but allows the errors in the statement and sorts it out! The browser Opera 3.60 does not complain initially, but only displays the first string: 'You typed: '. Whatever the rights or wrongs of these approaches, the statement is hard to read and causes a problem for at least one browser, so should be corrected. The problem is that the underline tag <U> has not been enclosed in either single or double quotes. Furthermore (and no browser could complain) the *input* variable is inside the last string, and so is correctly interpreted as the sequence ofcharacters *i n p u t*. Fixing the last line gives:

document.write("You typed: <U>" + input + "</U>")

The errors in the last line were syntactic errors — errors to do with the form (syntax) of the script. Other errors are semantic, where you write something which JavaScript cannot make sense of, or logical, where you simply do the wrong thing for the application you are trying to develop.

Working from the top of the script in Activity 4: line 1 declares *fore* and *back*, and line 2 assigns the document's background colour to *back*. However, *back* is not used in the rest of the script. It could be used in another script in the same document, but it could be declared and initialised to the background colour in such as script. So, let's remove *back* from the variable declaration and remove the second line:

```
var fore
```

Line 3 is a mess. It has two assignments in it. You might not think so, but this is syntactically OK. What it means is that the rightmost assignment takes place first (*document.fgColor = fore*) and then its result is assigned to the next variable to the left — to *fore*. This type of assignment can make sense, but not here, because *fore* has no value — it is *undefined* (different to *null*) — and so its assignment to *document.fgColor* attempts to make that property undefined. But it stays as it was before. To figure this out you would have needed to insert alert or write method calls before and after the multiple assignment. Not only does line 3 contain a semantic error — an inadvertent JavaScript error — it contains a logical error which is not needed for the script to fulfil its purpose. The logical error is trying to change the foreground colour, whereas it looks like fore is to be used in a subsequent prompt, so it does need to be set to the foreground colour. Hence, we simplify line 3 to be:

```
fore = document.fgColor
```

Line 4 seems straightforward: it appears to be prompting the user for a new foreground colour, using the current foreground colour (in *fore*) as a default (albeit as HEX, rather than as a user-friendly name). But, *newFore* has not been declared. In fact, this is legal in JavaScript, but is not helpful to the human reader and so should be included in line 1. The same is true of input subsequently, so let's add it to line 1 as well:

```
var fore, newFore, input
```

Line 5 assigns the object remembered by *newFore* to the document's foreground colour. However, nonsense could have been input, or Cancel may have been clicked. Either way, *newFore* may not represent the foreground colour. Either reset *newFore* to be the background colour prior to the prompt (leaving the latter as it stands). Alternatively used *document.bgColor* in both lines and remove *newFore*. Let's do the former and complete the repaired script:

```
var fore, newFore,
input fore = document.fgColor
newFore = window.prompt("Foreground colour:",fore) document.fgColor =
newFore
newFore = document.fgColor
input = window.prompt("What do you want written out underlined in colour
document.write("You typed: <U>" + input + "</U>")
```

As you might have realised, there is nothing wrong with enclosing the final question mark in single quotes.

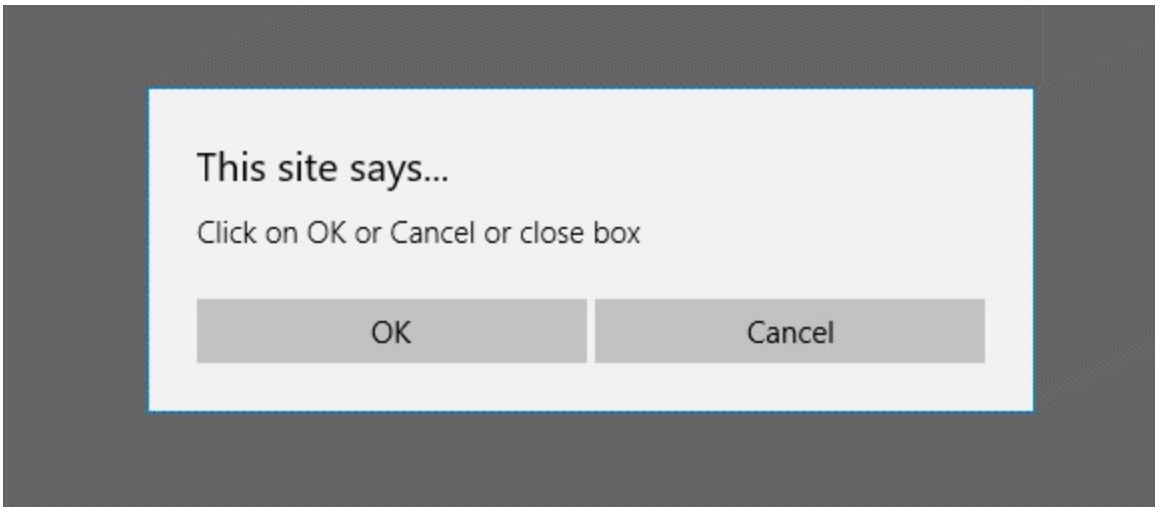
Note how long this activity took. Although the script was simple, a lot of thinking was needed to work out what was wrong with it. Be aware that you may have to spend a lot of time 'debugging' scripts like this.

11.5.9 Activity 6: The confirm method

All you need is to try out the method three times — one for each of the ways in which you can complete interacting with it: clicking OK, clicking Cancel, and closing its window. For example:

```
var confirmation
confirmation = window.confirm("Click on OK or Cancel or close box")
document.write("You clicked <B>" + confirmation + "</B>")
```

With Microsoft Edge this produces the following dialogue box:



Clicking OK assigns the value *true* to the variable *confirmation*, as verified in the document, below. Clicking on Cancel or closing the window of the dialogue box, returns *false*. You will use *true* and *false* frequently in JavaScript, but nor in this unit!

11.5.10 Activity 7: Changing the window status

Here is the entire body of the document:

```
<BODY>
<A HREF = "http://www.most-expensive-sellers.com" > Come to out cheap
on-line store
</A>
<SCRIPT>
window.status = 'Sale: all USB devices at 50% discount' </SCRIPT>
</BODY>
```

Ideally, if we are to change the status text, we want to vary it as the user moves around the window. We will see how to that when we discuss events in the next unit.

11.5.11 Activity 8: Semicolons to end statements

```
document.bgColor = "blue"; document.fgColor = "white";
document.write("This text should be white on a blue background.");
```

11.5.12 Activity 9: including separate JavaScript files

The HTML file would contain all of the HTML of the earlier version, but with the <SCRIPT> tag changed:

<!DOCTYPE html>
<!-- Copyright (c) 2015 UCT --> <html>
<BODY>
<SCRIPT SRC="foreColour.js"> </SCRIPT>
</BODY>

</html>
The file foreColour.js would then contain:
document.bgColor = "blue" document.fgColor = "white"
document.write("This text should be white on a blue background.")
One advantage of this style is that if a user was to save the source of a Web document, the included JavaScript file is not saved.

11.5.13 Activity 10: Opening a new Window

The new window if placed in the foreground and old document (with the URL of the file you use) is left in the background. The variable *document* still refers to the original document, now in the background window, hence that document has a blue background and contains new yellow text.

11.5.14 Answer to Review Question 1

JavaScript allows Web pages to be interactive, i.e. to part of a Web application.

11.5.15 Answer to Review Question 2

In the JavaScript object model, parts of an object's state are called 'properties'. A function that belongs to an object is called a method.

11.5.16 Answer to Review Question 3

When a user, for example, clicks on a hyperlink, the browser requests a document from a Web server. The server delivers it back to the user via his or her browser. The browser replaces the content of the current window with the newly rendered view of the document, that is, with the original document's HTML formatted. If the browser encounters JavaScript code, the instructions contained in the script are acted on by the browser.

11.5.17 Answer to Review Question 4

The JavaScript object *window* has methods *alert*. The *alert* method simply displays a string in a dialogue box and waits for the user to click OK (or close the dialogue box).

11.5.18 Answer to Review Question 5

An argument is a piece of information needed by a method. You supply an argument by enclosing it in parentheses, as in
document.write("Delivery date: ")
, in which case the argument is
"Delivery date: ".

11.5.19 Answer to Review Question 6

The following JavaScript code will display the string as required:
window.alert("Form ready for your application. Please proceed.")

11.5.20 Answer to Review Question 7

A variable is used to remember the result of evaluating some expression, typically to remember the result returned by a method.
A variable declared in one script in a document is available for use in all other scripts in the document.

11.5.21 Answer to Review Question 8

The following script would produce the required prompt.
window.prompt("WWW Guitar Auctions Inc. How much would you like to bid for guit

However, in reality some variables would be set to specific value and then used for in a more general prompt. For example, the company name may be held by a variable, the number of the item being sold may be held by a variable, and the minimum to be bid may be held:

var company = "WWW Guitar Auctions Inc."

var itemNo; minimumBid = 1
itemNo = 2873 //in reality something more complicated would set this
minimumBid = 100 //and this
window.prompt(company+ " How much would you like to bid for guitar #"+itemN

11.5.22 Answer to Review Question 9

You would use the following:
theDate = new Date(662989080980)

11.5.23 Answer to Review Question 10

Because no object name precedes it, this method must belong to the window object, and so could be written *window.close()*.

11.5.24 Answer to Review Question 11

The script resizes the window containing the document of which the script is part. It resizes to a square (hence the variable name *squareSize*). The script begins by prompting the user for the size of the square and suggests a size of 300.

By the way the 'odd' situations give rise to unpredictable behaviour in the browsers. Clicking Cancel sets *squareSize* to null and so the browser window may become very small, or remain unchanged — depending on the browser. The same happens with a negative number.