Chapter 16. JavaScript 3: Functions

Table of Contents

Objectives

At the end of this chapter you will be able to:

- Understand the importance of functions;
- Write HTML files using JavaScript functions;

14.1 Introduction

14.1.1 Introduction to JavaScript Functions

JavaScript functions are usually given a name, but since JavaScript functions are just objects in their own right, they can be stored in variables and object properties (see later unit). Functions are different from other objects in that they can be invoked (executed) with the use of a special operator *()*.

JavaScript provides many pre-written (*built-it*) functions, for example the function *write*, provided by the *document* object (see in a later unit how related functions can be stored inside objects; as we noted a few units ago, such functions are called methods).

An example of the *write* function being invoked is as follows:

```
document.write( "This message appears in the HTML document" );
```

An example of the alert function being invoked is as follows:

```
alert( "This message appears in an alert dialog" );
```

Some functions return a value. For example, mathematical functions perform calculations on the provided data and return numerical results. Other functions return true/false values, or text. Some functions return no value at all, but rather perform a side-effect; *write* is one such function whose side-effect is to send text to the HTML document. Functions frequently both perform a side-effect and return a value.

14.1.2 Uses of Functions

Functions can be used for many different purposes. In the past, JavaScript was used to create scrolling status bar messages for sites that want to give out important information that the user may miss while browsing the site (these proved to be largely irritating to the user, and have been very rarely used in the last few years). Displaying the date and time is also a common use of JavaScript. Image mouseovers, implemented using the HTML event system described in the previous chapter, are also widely used.

Functions are very useful when used in conjunction with HTML forms. Form entry can be validated, and the input of early parts of the forms might be used to invoke functions to automatically enter appropriate data in other parts of the forms.

To Do

Read up about JavaScripts Functions in your textbook.

14.2 Using Functions

14.2.1 Using built-in functions

The following lines illustrate the use of built-in functions:

```
document.write( "Hello" ); document.write(
Math.sqr( 2 ) );
document.write( "The bigger of 4 and 5 is : " + Math.bigger(4, 5) );
```

14.2.2 Using user-defined functions

You can define your own functions in the same file that they are invoked in, or in a different file which you can then load in a browser whenever you wish to use the function. Each of these situations are illustrated below.

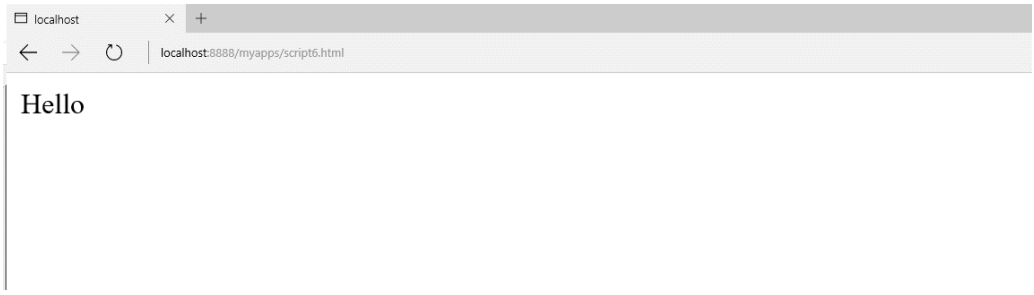
14.2.3 Defining and invoking a function in the same file

The following code defines and invokes a function named *displayHello*:

```
<HTML>
<SCRIPT>
////////////////////////////
/// define function here ///
//////////////////////////// function
displayHello()
{
document.write( "Hello" )
}
////////////////////////////
/// invoke function here ///
//////////////////////////// displayHello();
</SCRIPT>


</HTML>
```

The browser output when this HTML file is loaded is as follows:



14.2.4 Invoking a file defined in a different file

Some functions prove very useful; in order to use them in multiple Web pages they can be stored in a separate file. In the example below, the function *displayHello* has been defined in the file helloFunction.js. The HTML below uses two <*SCRIPT*> tags, one to load the function definition from helloFunction.js, and the second to invoke the function:

```
<SCRIPT SRC="helloFunction.js"></SCRIPT>
<SCRIPT> <!--
/// invoke function here /// displayHello();
</SCRIPT> -->
```

The contents of the file helloFunction.js is simply the JavaScript definition of the function:

```
/// define function here /// function displayHello()
{
document.write( "Hello" )
}
```

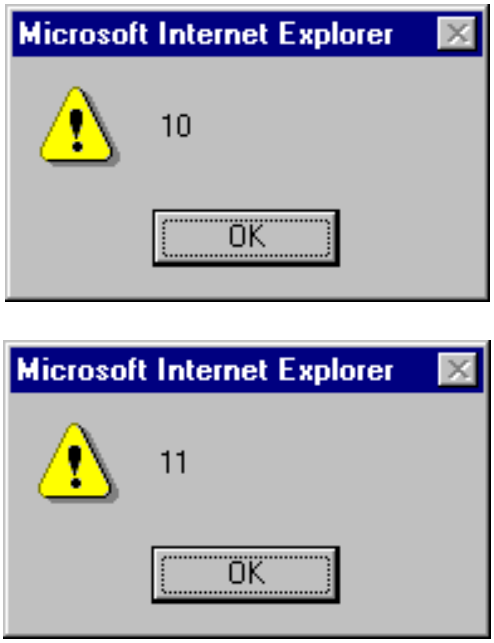
Notice that helloFunction.js is not an HTML file and does not contain any HTML tags. This is signified by choosing an appropriate file extension — the convention is to use the two-character extension ".js" for JavaScript files.

14.2.5 Executing code using 'eval'

The *eval* operator expects a String containing JavaScript as an argument, and will execute the String as if it where a JavaScript statement. The code below creates a String named *myStatements* and then executes the String using *eval*:

```
var myStatements = " var n = 10; alert( n ); n++; alert( n ) " ;
eval( myStatements );
```

The result of executing this code is the two alert dialogs:



14.3 Creating user-defined functions

14.3.1 A simple function to display the String “hello”

Let’s assume that we want to create a function that simply executes the following line:

```
document.write( "Hello" );
```

This function does not need any arguments, since it will do the same thing every time. The body of our function will be the single JavaScript statement above.

The table below illustrates a design for our function:

Name (optional)	
Arguments (optional)	

body	document.write("Hello")
Returns (optional)	

Depending on how we create the function, we may or may not need to name the function (in most cases it is useful to name functions).

14.3.2 Creating a function using function statements

The most convenient way to define a function is to use the *function* operator. The following example defines the *displayHello* function previously used:

```
function displayHello()
{
document.write( "Hello" );
}
```

The function operator requires the following:

```
function displayHello()         -- Function name followed by list
of arguments
{                                           -- Open Brace
document.write( "Hello" );   -- Sequence of JavaScript statements
forming the
}                                           -- Close Brace
```

As can be seen above, if there are no arguments an empty pair of parentheses () is written after the function name.

Our function design looks like the following:

Name (optional)	displayHello
Arguments (optional)	
body	document.write("Hello")
Returns (optional)	

14.3.3 Creating a function using the 'Function()' constructor

Another way to define a function is by using the *Function()* constructor. Recall that functions are themselves just objects, and that objects are created using constructor functions. *Function()* allows a function to be defined by passing a series of *Strings* to it. All but the last *String* lists the arguments to the function, while the last *String* defines the sequence of statements that form the function's body.

The *Function()* constructor returns a function which can be assigned to a variable of your choice. So, we can

now define displayHello() in this alternate way:

```
var displayHello = new Function( "document.write( 'Hello' );" );
```

Notice how there is no need for braces *{ }* to be used, since the body statements are contained in a String. Defining functions with *Function()* constructor is less convenient than using the function operator, but is extremely useful when dynamically creating functions, since the function arguments and body can easily be created as Strings.

Notice the single quotes around *'Hello*' — a double quoted *String* cannot appear inside a double quoted *String*. Having single quotes on the outside, and a double quoted *"Hello"* works just as well:

```
var displayHello = new Function( 'document.write( "Hello" );' );
```

In our above call to Function() there is only one *String*, since displayHello() requires no arguments. As before,

our function design looks like this:

Name (optional)	displayHello
Arguments (optional)	
body	document.write("Hello")
Returns (optional)	

14.3.4 Creating a function using function literals

Another third way to define a function is by using a function literal. This approach is very similar to using the function operator, except that the function is now nameless, although it can still be assigned to an arbitrary variable.

To define *displayHello* using a function literal, we would write the following:

```
var displayHello = function() { document.write( "Hello" ); }
```

Notice that function literals use a very similar syntax to function statements, and differ only in that the name of the function is not given.

Note

Although there are three different ways to define a function, in most situations you will find that using named functions defined with the function operator (the first technique described above) is easiest, although the other techniques all have their uses.

All the examples in the rest of this unit (and in most of the other units) define functions using the function operator.

14.4 Some simple functions

14.4.1 Mathematical functions

A JavaScript function to add two numbers:

```
function add()
{
  document.write( 5+5 );
}
add();
```

Name (optional)	add
Arguments (optional)	
body	document.write(5+5)
Returns (optional)	

And functions to perform various other arithmetical operations:

```
function minus()
{

  document.write("<p>" + (6-4) );

}
function times()
{

  document.write("<p>" + 6*4 );

}
add();
minus();
times()
```

Note: Function naming convention

You should always name your functions and variables with a lower case first letter, unless you are writing a function that will act as a constructor (see later unit). If your function name is made up of a number of words, start the second and subsequent words with an upper case letter to make the names more readable. Examples might include:

- function calculateTaxTotal()
- function changeImage()
- function setCircleRadius()
- function divide()

14.4.2 Functions that RETURN a value

Most mathematical functions do not display their results in an HTML document: they only calculate and return intermediate results for use by other functions.

For example, the code below uses the *Math.pow(value, power)* function to calculate the area of a circle. Also the *Math.round()* function is used to create *roundedArea*.

```
// calculate circle ara
var radius = 10;
var circleArea = 3.1415 * Math.pow( radius, 2 );
// round to 2 decimal places
var roundedArea = Math.round( circleArea * 100) / 100;
document.write( "<p> Area of circle with radius " + radius + "
has area of " + circleArea );
document.write( "<p> rounded area is " + roundedArea );
```

Area of circle with radius 10 has area of 314.15000000000003

rounded area is 314.15

14.4.3 Defining a function that returns a value

Creating a user-defined function that returns a value is straightforward: at some point in the function's execution a return statements needs to be executed. Typically, functions returning a value require arguments that will be used in the calculation of that value.

For example we might wish to define a function *addVAT(total)* which adds 14% to a total, and returns this new value. The specification for our function might appear as follows:

Name (optional)	addVAT
Arguments (optional)	total
body	return (total * 1.14)
Returns (optional)	Value representing 14% VAT added to total

```
function addVAT( total )
{
return (total * 1.175);
}
var goodsTotal -=50;
writeln(" total before tax = " + goodsTotal );
var newTotal = addVAT(goodsTotal);
writeln("<p> total with tax added " + newTotal );
```

As can be seen, to make a function return a value we include a return statement after having done the appropriate calculation.

Arguments are named like variables (in fact they can be though of as a kind of local variable). So these named arguments can be referred to in the processing and calculation of the function.

14.4.4 A date Function

The function below displays the current date and time:

```
//function name
function today()
//begin function statements
{
//declare variable dayAndTime
// and initialise to become a new Date object
var dayAndTime = new Date()
//write date to browser page
document.write("The date is: " + dayAndTime)
//end function statements
}
//invoke function
today()
```

The above code includes many comments explaining how it functions. Without these comments the function and its invocation would look as follows:

```
function today()
{
//declare variable dayAndTime
// and initialise to become a new Date object
var dayAndTime = new Date()
//write date to browser page
document.write("The date is: " + dayAndTime)
}
today()
```

The browser output is as follows:

Note

The behaviour of the scripts may vary according to which Web browser you use.

14.4.5 The today function described

This function has been defined with:

```
function today()
{
...
}
```

The first statement declares a variable called *dayAndTime* initialised with a *Date* object. *Date* was discussed in previous units, and allows access to the current date and time.

Note

JavaScript is case sensitive. The variable *dayAndTime* is not the same a *dayandtime*.

It is a good idea to keep all your JavaScript in one case. Lowercase letters, except for the first letter of the second and later words, is the most appropriate choice (unless working with class names).

At this point, the function now has the date and time stored, but has not yet displayed it. The second statement does this:

```
document.write("The date is: " + dayAndTime)
```

In the second statement the *document* object refers to the primary Web browser window. The method *write* allows content to be written to the browser.

The output is determined in this particular instance by:

```
("The date is: " + dayAndTime)
```

The value of variable *dayAndTime* is converted to text and concatenated (added) to the String *"The date is: "*.

This new String is sent to the write statement and displayed in the browser window.

Exercise 1

Open a new notepad document and write (or copy) the script for the *today* function. Save the document as "today.html" (or something similar) and load this HTML document into your Web browser. Your browser should look similar to the one in 14.4.4.

Your first task is to familiarise yourself with this function.

1. Change all uses of the function name today() to an alternative name. Save and reload your changed HTML file.

2. Change dayAndTime to an alternative name. Save and reload your changed HTML file.

3. Add the window.status = dayAndTime; statement. Remember to use the new name you've given the dayAndTime variable name.

What does this new statement do? Save and reload your changed HTML file. You can find some

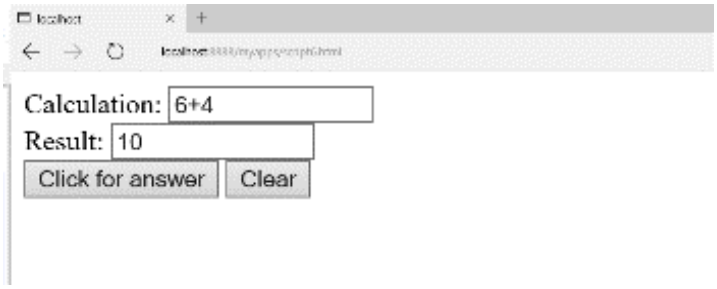
thoughts on this exercise at the end of the unit.

14.5 Mathematical Functions

14.5.1 A form for calculations

The form we shall develop

We shall progress in stages to a form providing a simple calculator as follows:



The above image shows an example of a single line text field with *4 * 7* in it. By clicking on the *Click for answer* button the result appears in the second (*result*) text field.

There are two important events that happen on such a form:

- The user clicking the "Click for answer button".
- The user clicking the "Clear" button.

A function that evaluates the calculation in the first text box and places the result in the second text box. The second of

these events can be more simply implemented with a form *reset* button.

The other events that occur will be the user typing in a calculation in the "Calculation:" text field. The browser will ensure that the user's input is stored in the appropriate form's text field.

The HTML for the form

The following HTML code displays the text, the text boxes and the two buttons:

```
<FORM>
Calculation: <INPUT TYPE=text NAME=expression SIZE=15><br>
Result: <INPUT TYPE=text NAME=answer SIZE=15><br>
<INPUT TYPE=button VALUE="Click for answer"
```

```
onClick="calculate(this.form)">
<INPUT TYPE=reset VALUE="Clear">
</FORM>
```

As you can see, when the first (Click for answer) button is clicked it will call the calculate() function, passing it the argument form. This argument refers to the form the button appears in.

The second button is a standard *reset* button called 'Clear'. When a '*reset*' button is clicked, the browser clears all text boxes in form.

There are two named text fields: expression and answer.

When the *Click for answer* button is clicked the JavaScript function *calculate()* is invoked, with the argument *this.form*. This argument refers to the form in which the function is being invoked from.

The JavaScript Function

The function is quite straightforward: it retrieves the expression the user has entered in the text field named expression, evaluates this expression and places the answer into the answer text field.

The function is passed an argument referring to the form where these buttons and fields are defined. The function specification can be written as follows:

Name (optional)	calculate
Arguments (optional)	theForm
body	`var result = eval( theForm.expression.value);` `theForm.answer.value = result;`
Returns (optional)	

We can refer to the value of a text field by:

```
[formname].[fieldname].value
```

We have named our function argument *theForm* — this argument will refer to whichever form the function has been called from.

As you can see, we are evaluating the contents of the expression form field as follows:

```
eval( theForm.expression.value )
```

The result of this expression is assigned to a variable called result:

```
var result = eval( theForm.expression.value);
```

The final statement assigns the result to the answer field.

```
theForm.answer.value = result;
```

The Full HTML file

The complete HTML file, including both function definition and HTML form, is as follows:

```
<HTML>  <HEAD> <SCRIPT>
function calc( theForm )
{
// evaluate the text field expression;
```

```
var result = eval( theForm.expression.value);
// put result into form field 'answer'
theForm.answer.value = result;
}
// </SCRIPT> </HEAD>
<BODY>
<FORM>
Calculation: <INPUT TYPE=text NAME=expression SIZE=15><br>
Result: <INPUT TYPE=text NAME=answer SIZE=15><br>
<INPUT TYPE=button VALUE="Click for answer"
onClick="calc(this.form)">
<INPUT TYPE=reset VALUE="Clear">
</FORM>
```

Note on eval statements

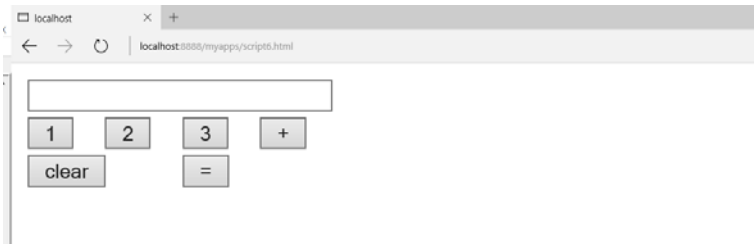
The eval statement will evaluate any JavaScript statement, not just those that perform mathematical calculations. For example, try entering *alert("Hello")* in the text box — when evaluated the browser will display the alert box.

We recommend validating the input to ensure that the expression is only mathematical before passing the entered text to eval.

14.5.2 A familiar calculator interface

The HTML for the form

Let us consider a simple version of the calculator that only provides the digits 1, 2 and 3 and the addition (+) operator:



We arrange the form in a table of three rows.

The first row of the table contains a text field to display the calculation and result:

```
<TR>
<TD colSpan=4><INPUT NAME=display size=30>
</TD>
</TR>
```

This first row uses a colspan of 4 to stretch over all the columns, and we name text field "display". The

second row contains four buttons (1,2,3 and +):

```
<TR>

<TD><INPUT TYPE=button VALUE=" 1 " onclick="append(this.form, 1)"
>
</TD>
<TD><INPUT TYPE=button VALUE=" 2 " onclick="append(this.form,
2)">
```

```
</TD>
<TD><INPUT TYPE=button VALUE=" 3 " onclick="append(this.form, 3)"
>
</TD>
<TD><INPUT TYPE=button VALUE=" + " onclick="append(this.form,
'+')" >
</TD>
</TR>
```

Each button has an *onClick* event handler that invokes a function *append()*. append() is passed two arguments: *this.form* refers to the form the calculated is defined in, and the second argument is the digit (or '+' character) to be appended to the *display* text field.

The third row of the table is composed of the *clear* and = buttons:

```
<TR>
<TD colSpan=2><INPUT TYPE=reset VALUE=" clear ">
</TD>
<TD colSpan=2><INPUT TYPE=button VALUE=" = "
onclick="calc(this.form)">
</TD>
</TR>
```

The *clear* button is another example of a *TYPE=reset* button. This button has been made to stretch over two columns.

The = button has an *onClick* event handler that invokes the *calc()* function. As per usual, we pass it the *this.form* form reference as an argument.

The JavaScript Functions

Our calculator form uses two functions: the *append()* function appends a digit or symbol to the display text field, and the *calc()* function evaluates the expression in the display text field, and replaces the text in the field with the calculated answer.

The *append()* function looks like this:

Name (optional)	append
Arguments (optional)	theForm, appString
body	`theForm.display.value += appStrin`
Returns (optional)	

The calc() function is specified as follows:

Name (optional)	calc
Arguments (optional)	theForm
body	`var result = eval( theForm.display.value );` `theForm.display.value = result;`
Returns (optional)	

The definition of the two functions follows:

```
function calc( theForm )
{
// evaluate the text field expression;
var result = eval( theForm.display.value );
```

```
// put result into form field 'answer'
theForm.display.value = result;
}

function append( theForm, appString )
{
theForm.display.value += appString
}
```

14.5.3 Some Function activities

Activity 1: Function to double a number

Create the specified function:

Name (optional)	displayDouble
Arguments (optional)	num
body	`var result = num * 2;` `document.write(num + " * 2 = " + result);`
Returns (optional)	

Invoke this function twice, first for the number 2, and a second time for the number 10.

Write a paragraph tag (<p>) to separate the displays in the browser window.

After invoking the function, the browser output should appear as follows:

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

Activity 2: Function to return four times a number

Create a function to return its argument multiplied by four. This function

should be invoked as follows:

```
document.write( "<p> 4 * 2 = " + fourTimes(2) );
document.write( "<p> 4 * 10 = " + fourTimes(10) );
```

Browser output should be the following when your function is invoked:

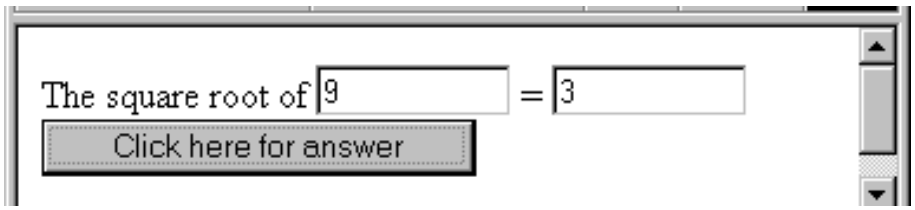
You can find a discussion of this activity at the end of the unit.

Activity 3: Decimal places function

Define a function to return its argument rounded to two decimal places. You can find a discussion of this activity at the end of the unit.

Activity 4: Square root calculator form

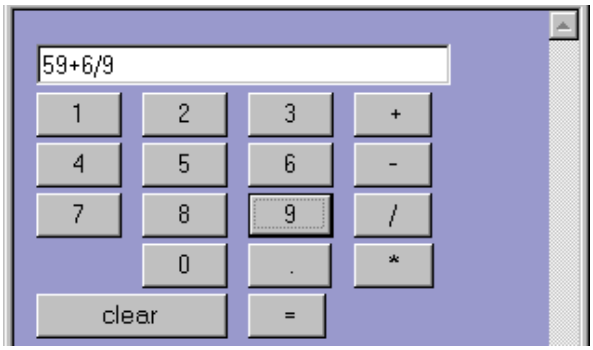
Create a form that calculates square roots. The browser should look as follows:



Hint: to calculate the square root of a number you can use *Math.sqrt()*. You can find a discussion of this activity at the end of the unit.

Activity 5: Completing the calculator form

Extend the file smallCalculator.html so that it is a complete calculator.



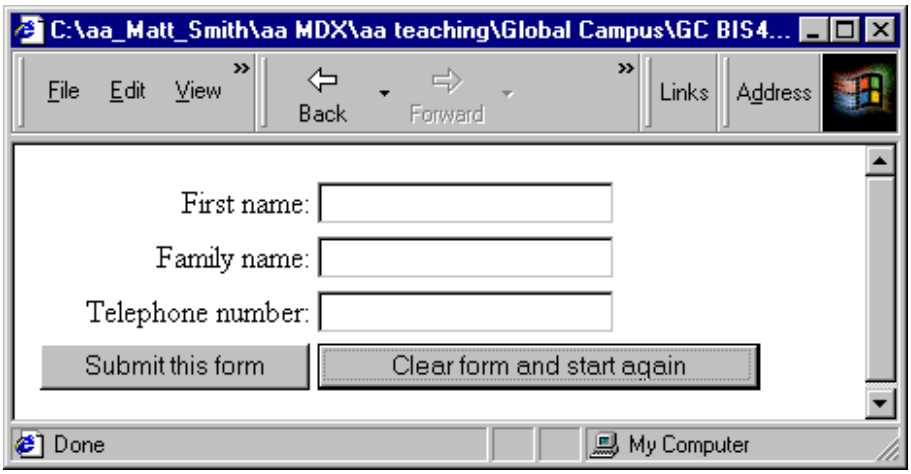
You can find a discussion of this activity at the end of the unit.

14.6 Form Validation

14.6.1 Testing for empty fields

An empty form field contains an empty String (i.e., "") as its value. Apart from seeing if the field's value is equal to the empty String, you can also test for an empty field by examining the length of the field's value. Since an empty field will have the empty String as its value, the length of the field's value will be zero.

Consider a page presenting the following form:



Let us assume that this form must have the first name and family name fields completed to be valid. The HTML

for the form is defined in a table. The form is named "order", and has an action to **post**
the input values — replace *your@email.address.here* with your own email address if you wish to try out this form yourself.

```
<FORM NAME=orderform METHOD="post" ACTION="mailto:your@email.address.here"
```

The form has been defined with an *onSubmit* event handler. This handler needs to return a Boolean (true/false) value, which, if false, will prevent the form from being submitted. Therefore we need to define a function called *validateOrderForm* that returns a Boolean value *true* if the form is correct, and *false* if it is invalid in some way.

14.6.2 The HTML defining the table

The first row of the table displays First Name and a text input field:

```
<tr>
<td align=right> First name: </td>
<td>
<INPUT TYPE="text" NAME="firstName" SIZE=20>
</td>
</tr>
```

The text *First Name* has been right aligned to improve layout and readability of the form. The input field has been named *firstName — NAME="firstName"*

The second row of the table displays *Family Name* and a text input field named *familyName*:

```
<tr>
<td align=right> Family name: </td>
<td>
<INPUT TYPE="text" NAME="familyName" SIZE=20>
</td>
</tr>
```

The third row of the table displays *Telephone number* and a text input field named *telephone*:

```
<tr>
<td align=right> Telephone number: </td>
<td>
<INPUT TYPE="text" NAME="telephone" SIZE=20>
</td>
</tr>
```

The fourth row of the table displays the two buttons:

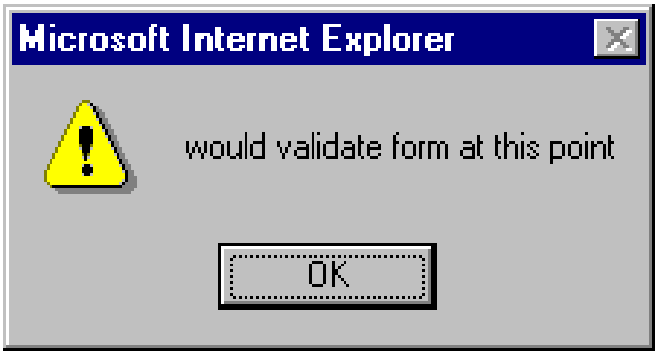
```
<tr>
<td>
<INPUT TYPE="submit" VALUE="Submit this form">
</td>
<td>
<INPUT TYPE="reset" VALUE="Clear form and start again">
</td>
</tr>
```

14.6.3 The JavaScript function to validate the form fields

We can test the above HTML using a dummy function that always returns false (so the form is never posted). Such a function could be written as:

```
function validateOrderForm()
{
alert( "would validate form at this point" );
// return Boolean valid data status return
false;
}
```

When we click the submit button we now see the alert dialogue appear:



To test if the *firstName* field is empty we can either compare the value of this field with an empty String:

```
orderform.firstName.value == ""
```

or we can test if the length of the value of this field is zero:

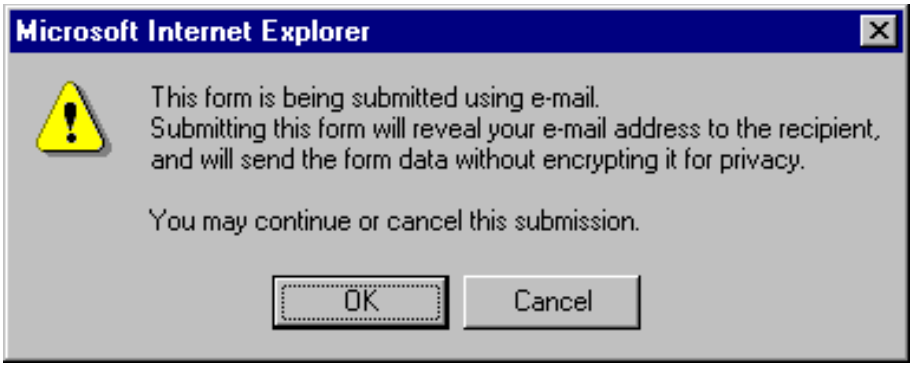
```
orderform.firstName.value.length == 0;
```

So if we wish our *validateOrderForm()* function to return true if the first name field has some value, and false otherwise we could write the function as follows:

```
function validate()
{
// at this point no invalid fields have been encountered var
fieldsValid = true;
// test field firstname
var firstNameValue = orderform.firstName.value; if
(firstNameValue.length == 0)
{
fieldsValid = false;
}
// return Boolean valid fields status return
fieldsValid;
}
```

As can be seen above, first the value of the *firstName* field is retrieved and assigned to a local variable called *firstNameValue*. Next, the length of this value is tested to see if it is zero. If it is, the Boolean variable *fieldsValid* is set to false to prevent the form from being submitted by making the function return false. Otherwise the value of this Boolean variable is left alone, and the function returns true.

If the first name field has had a value entered into, you may be informed that the form is about to be posted by the browser with a message such as the following:



14.6.4 Simplifying the code with a new function

The test we wrote for an empty text field is very useful for a page with many fields, so we can write a simple function called *isEmpty()* to do the test:

```
function isEmpty( fieldString )
{
if fieldString.length == 0 return true;
else
return false;
}
```

We can now rewrite the *validate()* function as follows:

```
function validate()
{
// at this point no invalid fields have been encountered var
fieldsValid = true;
// test field firstname
if isEmpty( orderform.firstName.value )
{
fieldsValid = false;
}
// return Boolean valid fields status return
fieldsValid;
}
```

14.6.5 Improving user interaction

While this works, the form is currently not very helpful to the user: if they click on the submit button and the first name field empty, nothing happens. At the very least the form should inform the user as to why it is not being submitted. This is easily solved by adding an alert statement, as in the following revised function definition:

```
function validate()
{
// at this point no invalid fields have been encountered var
fieldsValid = true;
// test field firstName

if ( isEmpty( orderform.firstName.value ) )
{
alert( "First name must have a value - form not submitted" ); fieldsValid =
false;
}
// return Boolean valid fields status return
fieldsValid;
}
```

Now if the submit button is pressed and the *firstName* field is empty, the form is not submitted, and the user is presented with the following dialog:

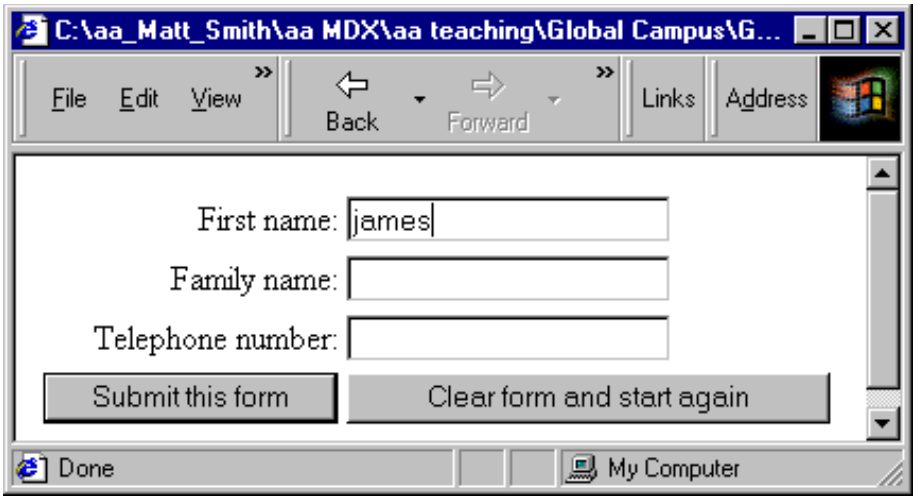
14.6.6 Validation of multiple fields

We can now extend our validate function to test the family name field as well:

```
function validate()
{
// at this point no invalid fields have been encountered var
fieldsValid = true;
// test field firstName
```

```
if ( isEmpty( orderform.firstName.value ) )
{
alert( "First name must have a value - form not submitted" ); fieldsValid =
false;
}
// test field familyName
if ( isEmpty( orderform.familyName.value ) )
{
alert( "Family name must have a value - form not submitted" ); fieldsValid =
false;
}
// return Boolean valid fields status return
fieldsValid;
}
```

If the user attempts to submit the form with an empty family name:



they will be presented with the following alert dialog and the form will not be posted:

14.7 Testing for numeric fields

Let us assume that only digits are permitted for the telephone number field (with no spaces, or dashes or parentheses for now). Some examples of valid telephone numbers are:

```
44181362500
002356487
56478303
```

Any entry that contains non-numeric values should be considered invalid; the user should be informed of this, and the form should not be submitted.

We can create a useful function *notNumeric()* that returns true if an argument passed to it is not a number. To write the function we can make use of the special value returned by JavaScript when the result of an attempt to perform a numeric

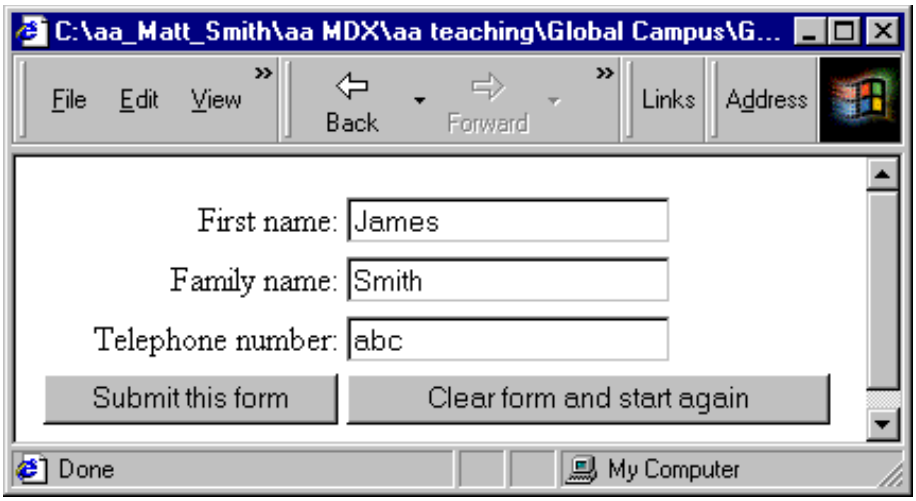
expression is not a number: JavaScript evaluates such expressions to the String *"NaN"*. Our function can be written as follows:

```
function notNumeric( fieldString )
{
if ( String(fieldString * 1) == "NaN") return true;
else
return false;
}
```

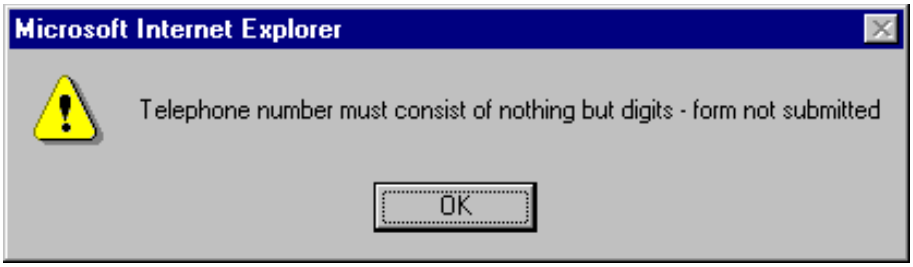
We can use this function to extend the *validate()* function to now only return true if the telephone number consists of digits:

```
function validate()
{
// at this point no invalid fields have been encountered var
fieldsValid = true;
// test field firstName
if ( isEmpty( orderform.firstName.value ) )
{
alert( "First name must have a value - form not submitted" ); fieldsValid
= false;
}
// test field familyName
if ( isEmpty( orderform.familyName.value ) )
{
alert( "Family name must have a value - form not submitted" );
fieldsValid = false;
}
// test field telephone
if ( notNumeric( orderform.telephone.value ) )
{
alert( "Telephone number must consist of nothing but digits - form not
submitted" );
fieldsValid = false;
}
// return Boolean valid fields status return
fieldsValid;
}
```

So if a telephone number of "abc" is entered:

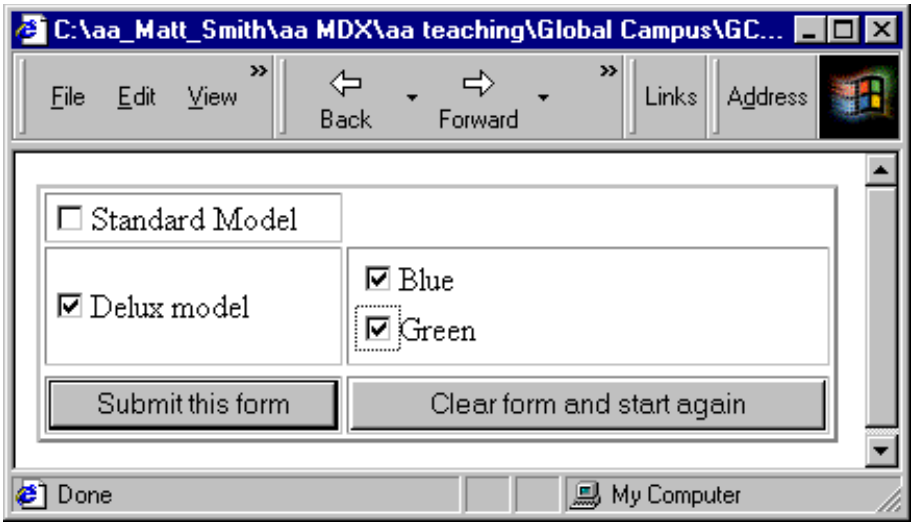


the browser will display the following alert window (and not post the form):



14.8 Testing for invalid field combination

Complex forms might require that only certain combinations of values/check boxes and so on be valid. The form shown below has an invalid combination of choices, and this can be picked up by a validation function. The form's submission can be cancelled and the user alerted to the problem.



On attempting to submit the form, the user is shown the following message:

The form defines three rows. The first row displays the Standard Model check box. The second row has two columns: the first contains the Deluxe model checkbox, and the second contains two colour check boxes (in a table of their own). The final row offers the submit and clear buttons as usual:

```
<FORM NAME=modelform METHOD="post"
ACTION="mailto:put.your@email.address.

<table border=2>
<tr>
<td>
<INPUT TYPE="checkbox" NAME="standard" value="Standard
model">Standard Model
</td>
</tr>
<tr>
```

```
<td>
<INPUT TYPE="checkbox" NAME="deluxe" value="Deluxe model">Deluxe model
</td>
<td>

<table border=0>
```

```
<tr>
<td>
<INPUT TYPE="checkbox" NAME="blue">Blue
</td>
<tr>
<td>
<INPUT TYPE="checkbox" NAME="green">Green
</td>
<tr>
</table>
<tr>
<tr>
<td>
<INPUT TYPE="submit" VALUE="Submit this form">
</td>
<td>
<INPUT TYPE="reset" VALUE="Clear form and start
again">
</td>
</tr>
</table>
</FORM>
```

The *validate()* function tests for a number of invalid conditions. The first test is made to see if both the standard and deluxe models have been chosen:

```
if( modelform.standard.checked &&
modelform.deluxe.checked )
{
alert( "Please choose either standard or deluxe (not
both) - form not submitted" );
fieldsValid = false;
}
```

The next two tests examine if a colour has been chosen with the standard model (this is not permitted):

```
if( modelform.standard.checked &&
modelform.blue.checked )
{
alert( "Sorry - colour choices are only possible with deluxe
cars - form fieldsValid = false;
}
if( modelform.standard.checked &&
modelform.green.checked )
{
alert( "Sorry - colour choices are only possible with deluxe
cars - form fieldsValid = false;
}
```

The final test ensures that only a single colour has been selected:

```
if( modelform.blue.checked &&
modelform.green.checked )
{
alert( "Please either blue or green (not both) - form not
submitted" ); fieldsValid = false;
```

```
}
```

14.9 The remaining activities

14.9.1 Activity 6: Completing the colour model form

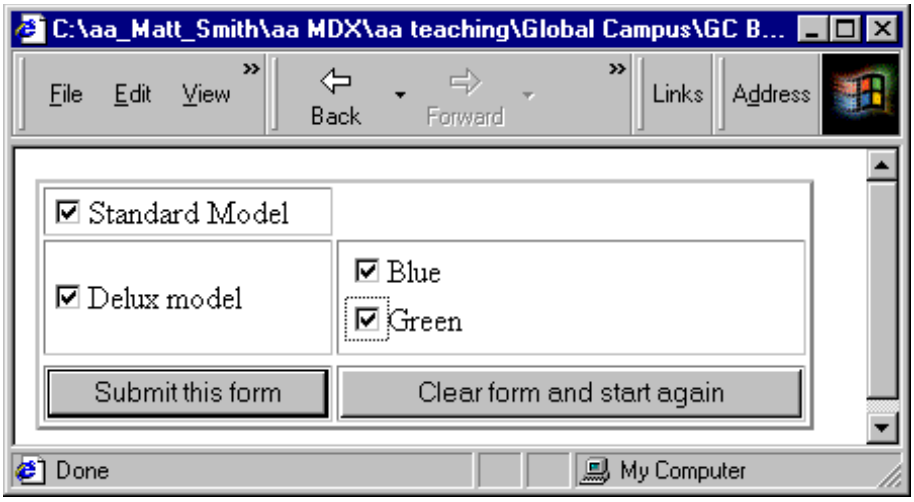
Extend the file form5.html so that it does not permit the form to be submitted when neither a standard nor a deluxe model have been selected.

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

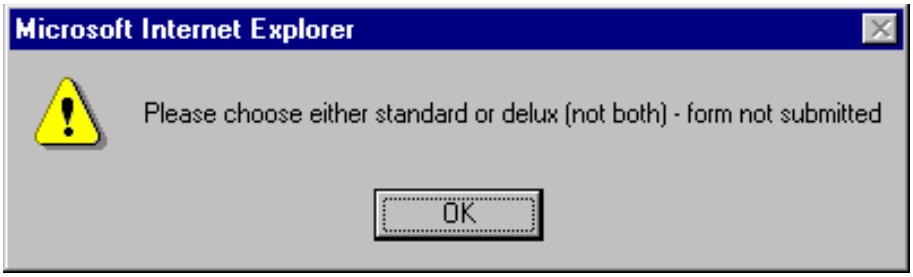
14.9.2 Activity 7: Avoiding Multiple Messages

It can be very annoying to users to receive many different error messages from the same form (i.e. three or four alerts all appearing one after the other).

For example, in the previous colour model form, if the screen were as follows:



the user would have to respond to the following sequence of alerts:



Amend the code so that an alert is only displayed for the first error encountered.

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

14.10 Review Questions

14.10.1 Review Question 1

What are the names of functions in the code below:

```
function displayMessage( message )
{
document.write( "<p> Message was: " + message );
}
displayMessage( "Hello" );
displayMessage( Math.sqrt( 25) );
```

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

14.10.2 Review Question 2

Write a specification for a function triangleArea to calculate and return the area of a triangle, for a provided height and width.

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

14.10.3 Review Question 3

What code would you need to create the following user-defined function:

Name (optional)	multiply
Arguments (optional)	n1, n2
body	`var result = n1 * n2;` `return result;`
Returns (optional)	Returns the result of multiplying the two given numbers

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

14.10.4 Review Question 4

What value is displayed when the following minus() function is invoked?

```
function minus()
{
document.write( 5 - 6 - 7)
}
minus();
```

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

14.11 Discussion Topic

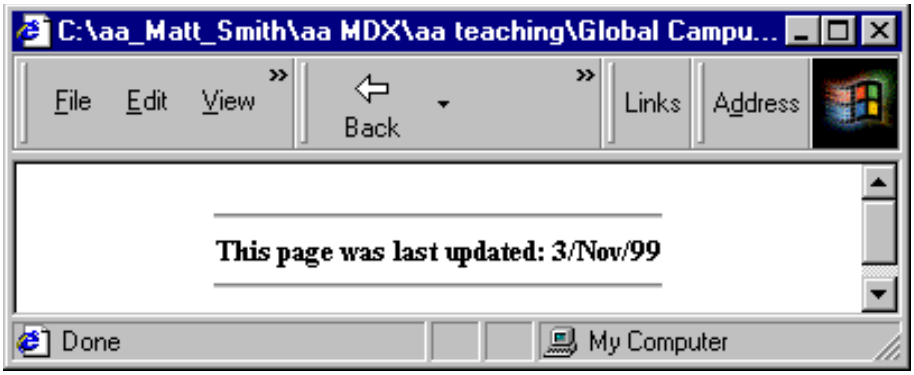
Generally, the definition of functions using the Function constructor and function literals is unnecessary since all functions can be defined using simple function statements such as the following:

```
function myFunction( arg1, arg2 )
{
// body of function
// optional function RETURN statement
}
```

You can find some thoughts on this discussion topic at the end of the unit.

14.12 Extension: More complex functions

A good example of a function that uses more complex JavaScript features, such as arrays and objects (see later unit), is a function to perform the standard task of displaying the date which a Web page was last updated. For example a 'banner' can be displayed at the end of each document as follows:



The code to create such output is as follows:

```
<HTML>  <SCRIPT> <!--
//function called update()
function update()
{
//declare a variable called
// (Modified to equal date of last save)
var modified = document.lastModified;
var months = new
Array("Jan","Feb","Mar","Apr","May","June","July","Aug",
"Sept","Oct","Nov","Dec");
//declare a variable called ModDate to equal last modified
date
var modDate = new Date( modified );
//write string of html formatting
document.write('<center><hr width=200><font size=2><b>');
document.write('This page was last updated: ');
//write day
document.write( modDate.getDate() + '/');
//write month
document.write( months[ modDate.getMonth() ] + '/');
//write year
document.write(modDate.getYear());
//write string of html formatting
document.write('</b></font><br><hr width=200></center>');
}
//invoke function
update()
// --> </SCRIPT> </HTML>
```

You may wish to examine this function now, and perhaps revisit it after working through the arrays and objects unit later in this module.

14.13 Discussions and Answers

14.13.1 Discussion of Exercise 1

The browser output now has the details of whatever day and time you opened the document in the status bar.

14.13.2 Discussion of Activity 1

We can define this function easiest using a function statement:

```
function displayDouble( num )
{
var result = num * 2;
document.write( num + " * 2 = " + result );
}
```

The function can be invoked with statements such as:

```
displayDouble( 2 );
```

```
document.write("<p>");
displayDouble( 10 );
```

14.13.3 Discussion of Activity 2

We can define this function easiest using a function statement:

```
function fourTimes ( num )
{
var result = num * 4;
return result;
}
```

14.13.4 Discussion of Activity 3

The simplest solution to this is to multiple the number by 100, round this value, then divide by 100 again.

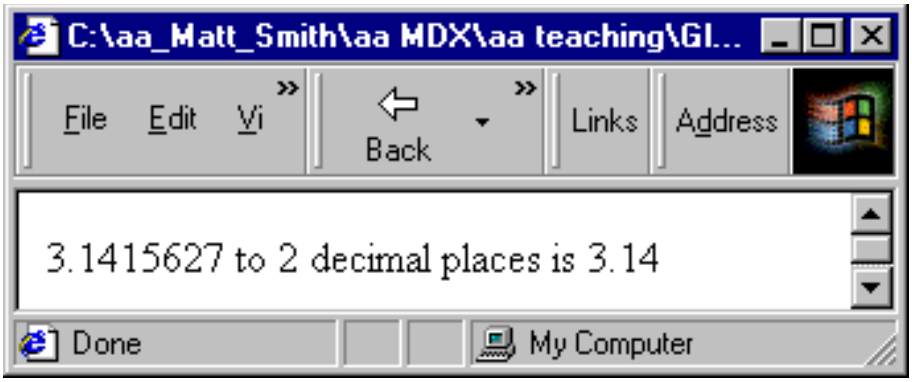
The function is as follows:

```
function twoDP( num )
{
var rouded = Math.round( num * 100 );
return rounded / 100;
}
```

An example of the function being invoked is as follows:

```
document.write(" 3.1415627 to 2 decimal places is " + twoDP(
3.1415627 ) );
```

The browser output should appear as follows:



14.13.5 Discussion of Activity 4

The code for the form should look similar to the following:

```
<form name="form2">
The square root of
```

```
<INPUT TYPE="text" NAME="number" VALUE="" SIZE=10>
= <INPUT TYPE="text" NAME="square" VALUE="" SIZE=10><br>
<input type="button" value="Click here for answer"
onClick="display()">
</form>
```

The code for the function should look similar to the following:

```
function display( square )
{
var num = document.form2.number.value;
document.form2.square.value = Math.sqrt( num );
}
```

14.13.6 Discussion of Activity 5

This is straightforward — all that needs to be done is to add another three rows of buttons to the form (with appropriate onClick event handlers). The existing functions can be left unchanged.

The extra HTML lines are:

```
<TR>
<TD><INPUT TYPE=button VALUE=" 4 " onclick="put(this.form,
4)"></TD>
<TD><INPUT TYPE=button VALUE=" 5 " onclick="put(this.form,
5)"></TD>
<TD><INPUT TYPE=button VALUE=" 6 " onclick="put(this.form,
6)"></TD>
<TD><INPUT TYPE=button VALUE=" - " onclick="put(this.form, '-
')"></TD>
</TR>
<TR>
<TD><INPUT TYPE=button VALUE=" 7 " onclick="put(this.form,
7)"></TD>
<TD><INPUT TYPE=button VALUE=" 8 " onclick="put(this.form,
8)"></TD>
<TD><INPUT TYPE=button VALUE=" 9 " onclick="put(this.form,
9)"></TD>
<TD><INPUT TYPE=button VALUE=" / " onclick="put(this.form,
'/')"></TD>
</TR>
```

14.13.7 Discussion of Activity 6

We need to add a new section to the validate() function, testing to see if neither are selected:

```
if( !modelform.standard.checked && !modelform.deluxe.checked )
{
alert( "You must choose either standard or deluxe - form not
submitted" ); fieldsValid = false;
}
```

Note the use of the exclamation mark ! — this is a *logical not* in JavaScript.

14.13.8 Discussion of Activity 7

One solution is to only display an alert for an invalid field if all previous fields have been valid. For example, we

could amend the testing of the multiple colour fields to the following:

```
if( modelform.blue.checked && modelform.green.checked )
{
if (fieldsValid)
alert( "Please either blue or green (not both) - form not submitted" );
fieldsValid = false;
}
```

The same approach needs to be taken for all but the first invalid field.

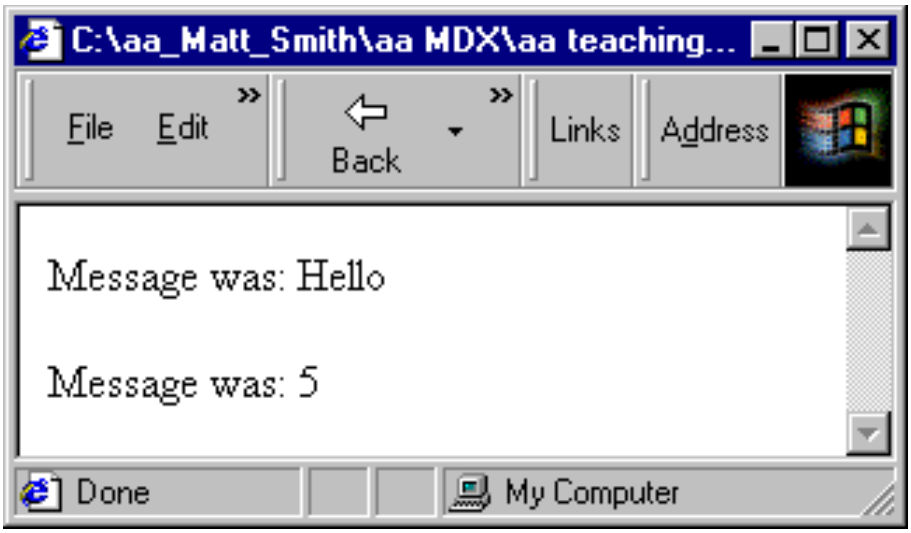
14.13.9 Discussion of Review Question 1

There are three functions referred to in this code:

1. The user-defined function *displayMessage()*
2. The built-in *write()* function (part of object *document* — see later unit)
3. The built-in *sqrt()* function (part of class *Math* — see later unit)

Generally, wherever you see parentheses, either function arguments are being defined, or a function is being invoked.

The browser output for the above code is:



14.13.10 Discussion of Review Question 2

This function should be named triangleArea.

triangleArea takes two arguments, which we shall call *height* and *width*.

The function needs to calculate the are of a triangle (1/2 * (*width* * *height*)). We can write a statement that assigns the result of this calculation into a variable called area:

```
var area = 0.5 * (height * width); The
function is to return the area: return
area;
```

The specification looks as follows:

Name (optional)	triangleArea
Arguments (optional)	height, width
body	`var area = 0.5 * (height * width) ;` `return area;`
Returns (optional)	Returns triangle area (0.5 * (height * width))

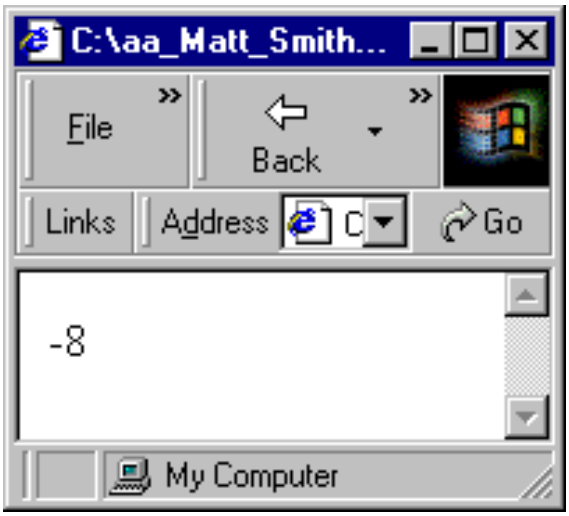
```
function triangleArea
{
var area = 0.5 (heigth * width) return area;
}
```

14.13.11 Discussion of Review Question 3

This function is most easily created using a function statement as follows:

```
function multiply( n1, n2 )
{
var result = n1 * n2; return result;
}
```

14.13.12 Discussion of Review Question 4



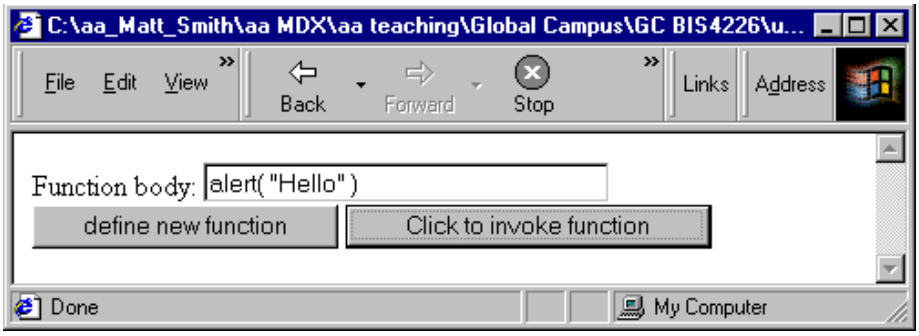
The function is invoked, the expression inside *write()* is evaluated to *-8*, and then the document.write() function is executed adding -8 to to the browser window.

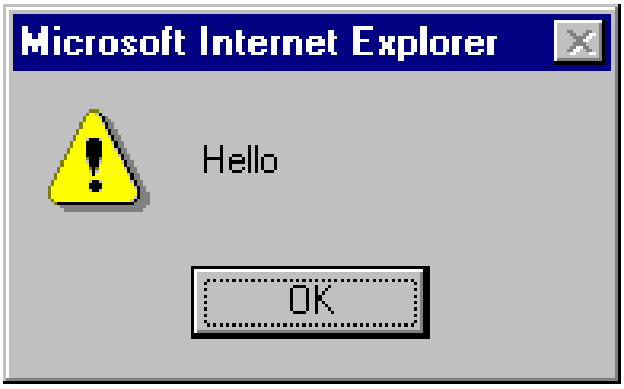
14.13.13 Thoughts on Discussion Topic

While in most cases it is convenient to define functions using function statements, there are situations when function literals and the Function constructor offer advantages.

For example, it is possible to have collections of functions that do not require names (see later unit on arrays and objects), in which case it is frequently convenient never to have to name them.

The Function constructor offers far more power in defining functions, since it defines the arguments and body of a function using Strings, and Strings are very easy to manipulate in JavaScript. This makes it possible to have the script itself create new functions as they are needed. Consider the browser output below:





This output is possible using a String entry from the user to define a function that can then be invoked through a button press. The code for the above is as follows:

```
<HTML> <HEAD> <SCRIPT> <!--
function userFunction()
{
alert( "no function yet defined" )
}

function buildFunction( theForm )
{


// DEFINE new function and replace existing 'userFunction'
userFunction = new Function( theForm.functionbody.value );
}

// --> </SCRIPT> </HEAD>
<BODY>
<FORM>

Function body: <INPUT TYPE=text NAME=functionbody SIZE=30><br>

<INPUT TYPE=button VALUE="define new function"

onClick="buildFunction( this.form )">
<INPUT TYPE=button VALUE="Click to invoke function"
onClick="userFunction()">
</FORM>
</BODY>
</HTML>
```

Although this particular page could have been more simply implemented using eval, it illustrates the flexibility of the Function constructor approach to function definition — a script can be written to define functions using statements not known when the script was written.