A SunCam online continuing education course

Web-Based Programming For Engineers - Part 3

by

Kwabena Ofosu, Ph.D., P.E., PTOE

Web-Based Programming For Engineers – Part 3
A SunCam online continuing education course

Abstract

The objective of this course series is to present web-based computer programming to engineers. Engineers generally learn a conventional computer programming language such as *FORTRAN*, *Pascal*, *C++*, etc. Since the advent of the internet and the World Wide Web, web browsers such as *Internet Explorer*, *Mozilla Firefox*, *Google Chrome*, etc. have built-in capabilities to interpret and implement programmed instructions written in a class of programming languages called scripting languages. Web-based programming involves writing codes, called scripts, in a scripting language. The scripts are embedded in the structure of web pages. Unlike conventional general purpose programming languages, web-based programming does not require any special software to be installed. The scripts are interpreted and implemented directly by the web browser. Web-based programming is an increasingly relevant and advantageous tool for engineers competing in the global marketplace in the age of the internet and the World Wide Web. Once uploaded to the World Wide Web, web-based applications are immediately exposed to a global audience.

This course is the final part of a series on web-based programming. This course presents topics on the *JavaScript* scripting language. This course uses screenshots and an easily readable click-by-click narrative that engages participants as they proceed through the topics. This course starts with an overview of the branching and looping structures in *JavaScript*, followed by an in-depth presentation of *JavaScript* objects. Techniques to manipulate the web browser as well as handling of errors are also presented. Examples from various fields are presented to illustrate the application of the fundamental concepts in real world situations. On completion of this course, participants will be able to create fully functional interactive web pages and web applications that can be used to input and output data, as well as run complex algorithms. On completion of this course participants will be able to identify professional situations in which applying web-based programming will be of great benefit to them in their fields of specialty and to their organizations.

There are no required pre-requisites for this course. However, it will be helpful to understand the basics of creating web pages as well the fundamentals of scripting languages as presented in the earlier parts of this course series.

Web-Based Programming For Engineers – Part 3
A SunCam online continuing education course

TABLE OF CONTENTS

List of Tables

A SunCam online continuing education course

1. CONDITIONAL STATEMENTS

1.1 Definition

A conditional statement is a feature of a scripting language that executes different instructions (lines of code) based on whether some **condition** is met. Conditional statements enable the programmer to control the way an application interacts with the user. Conditional statements are often referred to as **branching** as they provide a means for a program to branch off in some direction or the other as some condition(s) is checked for and met, and the program then proceeds in the relevant direction(s).

1.2 *If* Statement

The simplest conditional statement is the *if* statement. If a specified condition is met, a block of code will be executed. The *if* statement is of the structure:

```
if (condition)
{
        Run this code
}
```

The *if*...*else* statement is used to specify a block of code to run if the condition is met, and another block of code to run if the condition is not met. The syntax is as follows

```
if (condition)
{
        Run this code
}
else
{
        Run that code
}
```


For more than two conditions, the *if…else if* statement is used as follows:

```
if (condition 1)
{
        Run code 1
}
else if (condition 2)
{
        Run code 2
}
else if (condition 3)
{
        Run code 3
}
:
:
else if (condition n-1)
{
        Run code n-1
}
else
{
        If none of the above apply, Run code n
}
```

1.3 Logical Operators

The *condition* in the condition statement is a **logical expression** where a **logical operator** (also called a **Boolean operator**) is applied to compare, evaluate, or check that the inputs (called **operands**) meet the specified condition and give a result of "true", based upon which the relevant block of code will execute. Examples of logical operators supported in *JavaScript* are shown in Table 1.1.

Table 1: Logical operators

Operator	Description	Example
= =	Checks if the values of two operands are equal or not. If true, then the condition is "true", otherwise it is "false".	*if (X = = Y)* *{...* *}*
!=	Checks if the values of two operands are not equal. If the values are not equal, then the condition is "true", otherwise it is "false".	*else if (p != q)* *{ ...* *}*
>	Checks if the value of the left operand is greater than the value of the right operand. If true, then the condition is "true", otherwise it is "false".	*if (m > n)* *{...* *}*
<	Checks if the value of the left operand is less than the value of the right operand. If true, then the condition is "true", otherwise it is "false".	*else if (x < y)* *{...* *}*
> =	Checks if the value of the left operand is greater than or equal to the value of the right operand. If true, then the condition is "true", otherwise it is "false".	*if (a >= b)* *{...* *}*
<=	Checks if the value of the left operand is less than or equal to the value of the right operand. If true, then the condition is "true", otherwise it is "false".	*else if (q <= r)* *{...* *}*
= = =	Checks if the two operands are of equal value **and** equal type, or not. If true, then the condition is "true", otherwise it is "false".	*if (x === y)* *{...* *}*
! = =	Checks if the two operands are not of equal value **and** of different type, or not. If true, then the condition is "true", otherwise it is "false".	*if (x !== y)* *{...* *}*

1.4 Composite Conditional Expressions

Conditional expressions may be combined using the "and" (**&&**) and/ or "or" (||) operators to form a **composite conditional expression**.

The operators for composite conditions in *JavaScript* are presented in Table 2.

Table 2: Composite conditional operators

Operator	Description	Example
&&	"And"	*if (X > 50 && Y <20)* *{…* *}*
||	"Or"	*else if (p >20 || q>=15)* *{ …* *}*
!	"Not"	*if !(m = = n)* *{…* *}*

For example, consider a bank account that has been overdrawn. If another charge comes in and the bank pays it, the account goes further into the negative and is charged an overdraft penalty for that transaction. However, if a deposit comes in that partially clears the deficit, even though the account is still in the negative, the account is not charged an overdraft fee for that transaction. Therefore, using the negative sign for a charge transaction and positive sign for a deposit, the overdraft penalty fee is applied as follows:


```
if (balance < 0 && transaction < 0)
{
        newbalance = balance + transaction - fee ;
}
else if (balance < 0 && transaction > 0 )
{
        newbalance = balance + transaction ;
}
```

1.5 Nested Conditional Statements

A **nested conditional statement** is a conditional statement placed within another conditional statement. The bank account example can alternately be implemented using nested conditions as follows:

```
if (transaction < 0)
{
        if (balance < 0)
        {
                newbalance = balance + transaction – fee ;
        }
        else
        {
                // no fee applicable
        }
}

else
{
        // no applicable fee
        newbalance = balance + transaction ;
}
```


The *if* ...*else* and/or *if* ...*else if* syntax for each conditional statement must be complete on their own regardless of whether they are nested or not. For instance, in the above example, if the *else* or a bracket of the nested *if* ... *else* was omitted, the syntax would be incomplete and incorrect, and an error would occur. A common strategy to keep track of this, as demonstrated in the above example, is by typing the code such that the *if* ... *else* and brackets for a specific *if* statement are aligned vertically and that of any nested statements are offset laterally from the main statement in which they are nested. This is called **indenting** the code.

The choice, relevance, or advantage of nesting versus composite conditions must be determined by the programmer based on the specific objectives and requirements of the application.

1.6 Conditional Operator

The *JavaScript* conditional operator may be used to assign a value to a variable based on some condition. The syntax is as follows:

variable = (condition)? value1 : value2 ;

For example, a concrete specimen in a compression test must yield a result of 30 psi or more to "pass", otherwise it will "fail" and the contractor will not be paid.

testresult = (compstrength >= 30)?"Pass" : "Fail" ;

Using the conditional operator to implement this, if the variable *compstrength* has a value equal to or greater than 30, then the variable *testresult* will be assigned the value "Pass", otherwise it will be assigned the value "Fail".

1.7 Switch Statement

This is an alternate method to the *if* statement. It is advantageous to use when there is an excessive number of conditions and the *if* statement becomes cumbersome and difficult to follow and keep track of. The syntax is as follows:


```
switch(variablebeingchecked)
{
case value1:
        Run code 1 if variablebeingchecked = value1
        break ;
case value2:
        Run code 2 if variablebeingchecked = value2
        break ;
:
:
default:
        Run this code if none of the above cases applies
}
```

Each case *value* may consist of a value, or variable or logical expression, or composite logical expression, or any combination thereof. Starting from the top, if the value of the *variablebeingchecked* meets the condition of a case, then the relevant code will be run. Once the relevant code has run, the *break* will "kick us out" of the switch statement. Without the *break*, the browser will continue checking the next case in the switch statement and so on and so forth. The *default* case and its associated code applies if none of the above cases are met. For example,

```
switch(reading)
{
case  1:
        document.write("ON");
        break ;
case  -1:
        document.write("OFF");
        break ;
default:
        document.write("Device malfunction")
}
```

In other words if the *reading* value equals 1, the device is "ON". If the *reading* value is -1, the device is off. Any other reading implies we have a malfunction of the device.

2. MESSAGE BOXES

A message box, commonly called a **popup box** acts like a dialog box where a user can interact with the computer. Message boxes are used to alert or prompt the user. Message boxes are able to perform actions in response to what the user selects. Message boxes are often incorporated into conditional statements to control the flow of the code based on user's selection(s). There are three forms of the message box in *JavaScript*, namely, the **alert box**, the **confirm box**, and the **prompt box**.

2.1 Alert Box

In its simplest form, the general structure of the code for a message box uses the *alert( )* function as follows:

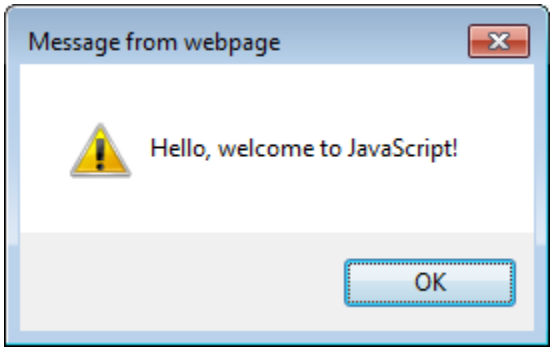
```
alert (some message for the user) ;
```

A message box created in this manner, referred to as an alert box, has an *OK* button only, which the user must click on to dismiss it and enable the rest of the code for the script to run.

For example,

```
alert("Hello, welcome to JavaScript");
```

Which yields,

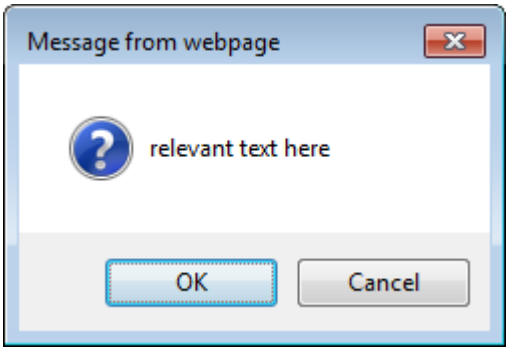


2.2 Confirm Box

The confirm box is used when the user is required to make a decision and select some option to proceed. The user must select "OK" or "Cancel" to continue. The syntax is

```
confirm("relevant text here") ;
```

which will yield:



Code must be written to determine events that will "fire" based on which button the user clicks on. Therefore confirm boxes are typically used in conjunction with *if* statements.

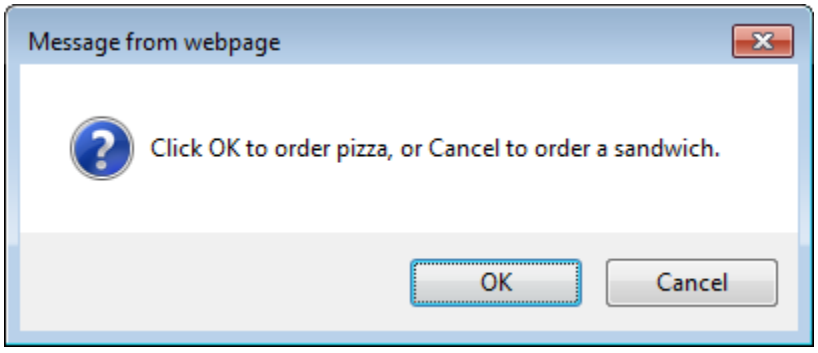
Consider the following example,

```
<script>

//the following line brings up the confirm box
var r=confirm("Click OK to order pizza, or Cancel to order a sandwich.");

if (r==true) //so if the user clicked on OK this block of code will execute
  {
        alert("You ordered a pizza!");
```

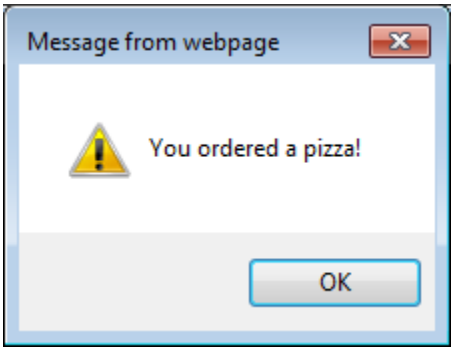


```
 }
else  //if the user clicked on Cancel then this block of code will execute
 {
        alert("You ordered a sandwich!");
 }

</script>
```

The confirm box appears as follows,



Clicking on *OK* yields:



Whereas clicking on *Cancel* yields:

2.3 Prompt Box

The prompt box is a popup box that enables the web page to retrieve input from the user. Prompt boxes are used to ask users questions, provide information, or communicate feedback to or from the user. The user clicks on the *OK* button to return input data to the web page, or the *Cancel* button to return a null value to the web page.

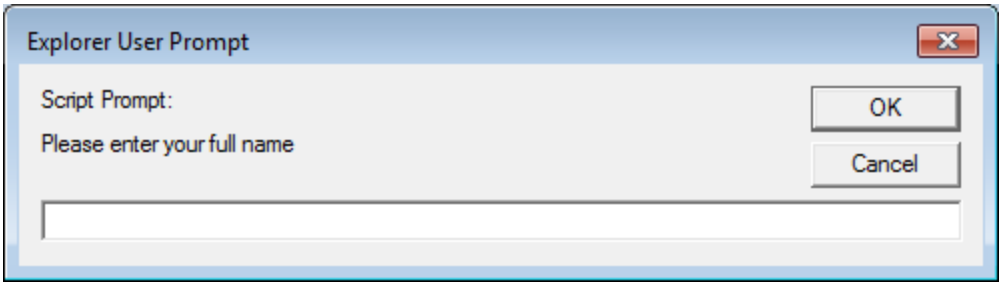
The syntax for the prompt box is:

prompt("relevant text here", "default text") ;

For example:

prompt("Enter your full name: ", "") ;

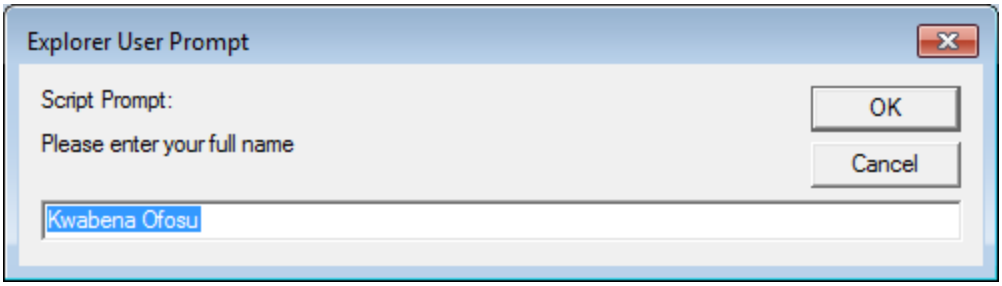
which yields



Or add your name as the default text in the text box, as follows:

prompt("Enter your full name: ", "Kwabena Ofosu") ;

This yields:



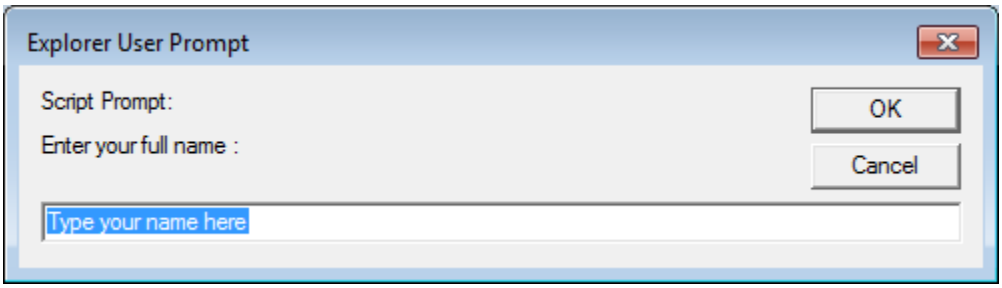
Code must be written to determine events that will execute once the input data has been supplied (or otherwise) and the *OK* button is clicked on. Prompt boxes are typically used in conjunction with *if* statements. Consider the following script,

```
<script>
var x;
//the following line brings up the prompt box and assigns its value to a variable
var r=prompt("Enter your full name : ", "Type your name here");
```

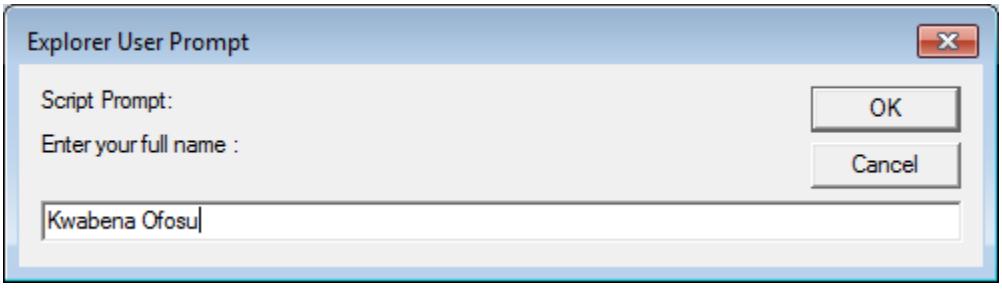


```
if (r!=null) //this is where we write the code for the OK button
              //on clicking OK, this block of code will execute
              //note the line break characters in the text
 {
      x = "Hello " + r  + ".\nThanks for visiting our website today.\nHave fun." ;
         alert(x);
 }
         // on clicking Cancel the prompt box dismisses and a null value is
         // assigned to the variable, and nothing else happens

</script>
```

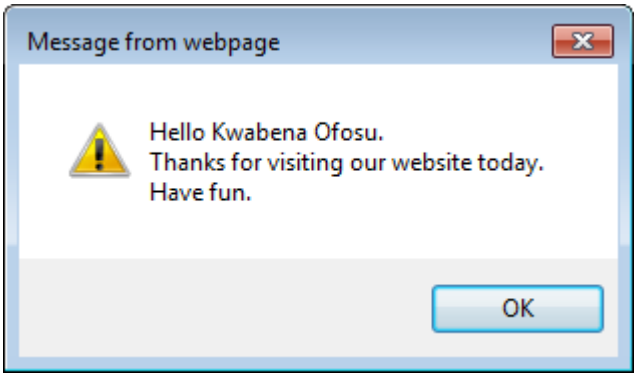
which yields:



Entering a name,

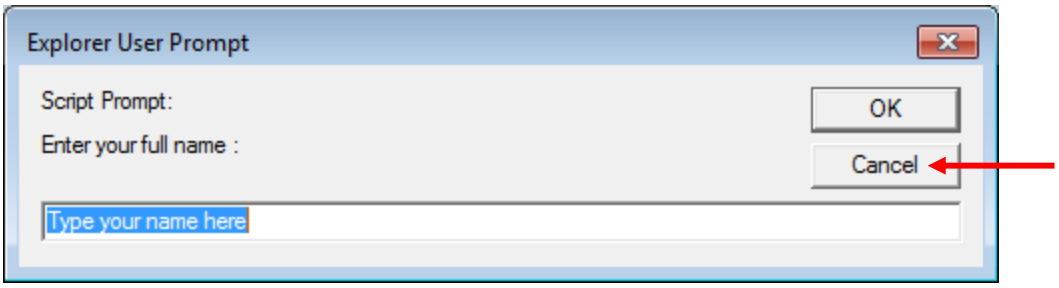


Click on *OK*.



Click *OK* to dismiss the alert box. The prompt box also dismisses as it has completed its code execution.

Repeat the process, but this time after entering a name (or without entering a name, or after clearing the text box), click on *Cancel*.



The prompt dismisses as expected.

2.4 Practicum #3

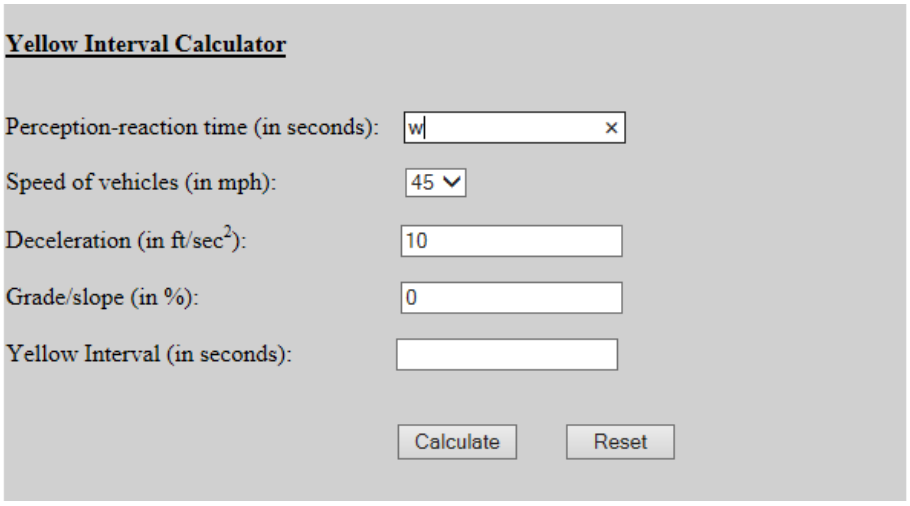
In this practicum we shall further develop the yellow time calculation application for the County Traffic Operations program that we designed in Practicum #1 and gave functionality in

Practicum #2 of this course series. Particularly, we shall add conditional statements and alert boxes to enforce the entry of admissible data for our calculations.

Solution

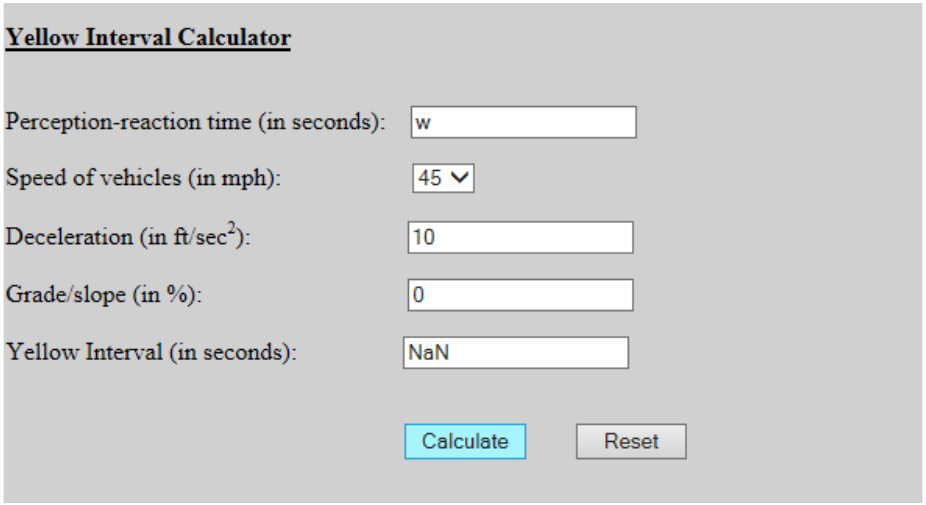
Open the application in your web browser.

Enter a non-numeric value in any one of the input boxes.



Click on *Calculate*. This yields *NaN* (not-a-number).

In other words we have entered inadmissible data for a valid numerical result to be calculated.



We can use the built-in function *isNaN( )* to determine whether an input value is *NaN* or not. If the test evaluates to "true" we shall throw an alert to the user to enter admissible data, and then terminate the calculation procedure.

Open your web page with *Notepad* (or your preferred text editor).
Scroll to the code for the function that calculates the yellow time, *fnYellowCalc( )*.
Add the conditional statement for the reaction time variable *reacTime* such that if the value assigned from the reaction time text box *txtReacTime* is not a number, the user is thrown an alert box telling them to enter a valid numeric value.
The function *return* call is then used to terminate the function after the user dismisses the text box.
If the user enters a valid numeric value then the *reacTime* variable is reassigned its value from the *parseFloat* function.

Practicum3 - Notepad

File Edit Format View Help

```
     Traffic Operations

  </title>


<script type="text/javascript">
//<script language="JavaScript">

  function fnYellowCalc()
        {
              //declare variables needed for the calculation
              var reacTime, speed, decel, gradient, yellowtime ;

              /*now assign the controls' values to the variables, identifying
              each control by its unique id */
              reacTime = document.getElementById("txtReacTime").value;
              speed = document.getElementById("optSpeed").value;
              decel = document.getElementById("txtDecel").value;
              gradient = document.getElementById("txtGrade").value;

              //check that the entry for reaction time is a valid numeric value
              //if not throw alert and stop the function
              if(isNaN(reacTime))
              {
              alert("Enter a numeric value for perception reaction time!");
              return;
              }
              else
              {
              //now ensure variable values are converted to numbers for calcs
              reacTime = parseFloat(reacTime);
              }

              //since user must select speed from combo box we wont
```


Repeat the conditional set up for the other input variables, deceleration, and gradient.
The user is restricted to selecting a speed from the combo box; therefore, checking and validating the speed entries may be omitted. This is a specific advantage of using combo boxes, list boxes, etc., when they are applicable.

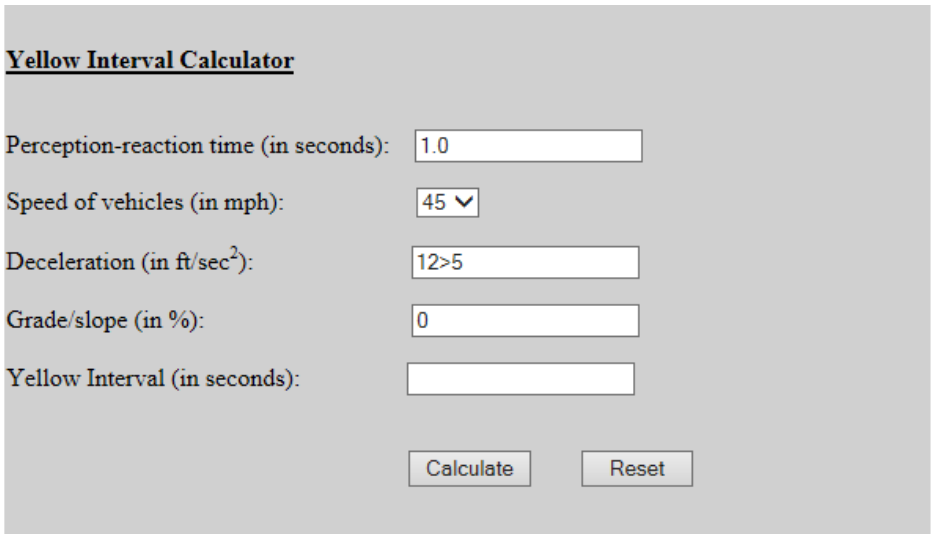
Practicum3 - Notepad

File Edit Format View Help

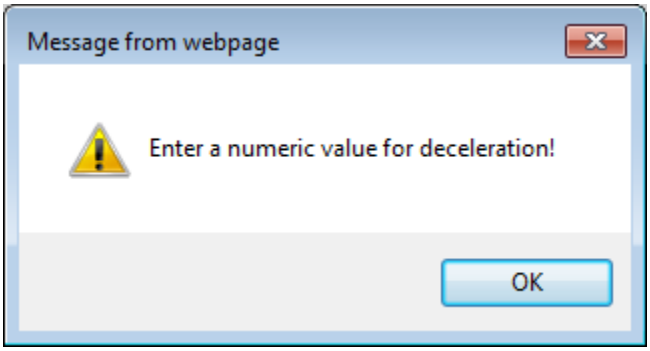
```
                //since user must select speed from combo box we wont
                //need to set up validation check
                speed = parseFloat(speed);

                //check that the entry for decelaration is a valid numeric value
                //if not throw alert and stop the function
                if(isNaN(decel))
                {
                alert("Enter a numeric value for deceleration!");
                return;
                }
                else
                {
                decel = parseFloat(decel);
                }

                //check that the entry for gradient is a valid numeric value
                //if not throw alert and stop the function
                if(isNaN(gradient))
                {
                alert("Enter a numeric value for gradient!");
                return;
                }
                else
                {
                gradient = parseFloat(gradient);
                }

                //calculate the yellow time using the ITE formula
                yellowtime = reacTime + 1.47*speed/(2*decel+32.2*gradient/100) ;

                //assign calculated value to the text box to display it
                frmYellowCalc.txtYellowInt.value = yellowtime ;

                return ;
        }


function fnAllredCalc()
```

After the validation checks, the yellow time calculation follows and stores a valid numerical result to the variable *yellowtime*. The *return* call ends the function, *fnYellowCalc( )*.
Save your file.
Open your file in your web browser to test it.

Enter an inadmissible value in the reaction time, deceleration, or gradient text boxes.

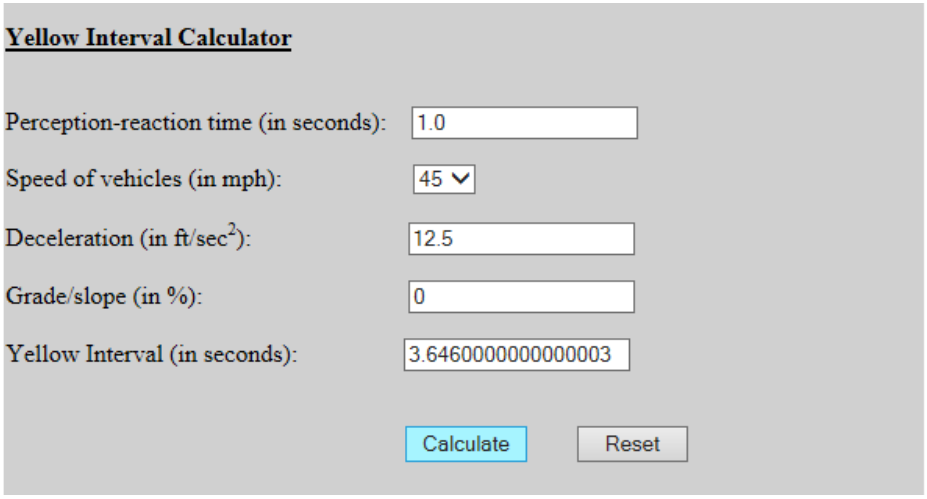


Press *Calculate*.



Dismiss the alert box.
Test the other text boxes.

Enter all valid inputs and press *Calculate*.



All inputs are checked and validated, and a valid result is calculated.
The test is a success.
For further practice, add relevant validation checks to the All-Red Clearance calculator.

3. LOOPING

Looping is a procedure in a scripting language that performs repetitive (iterative) tasks. The loop is a sequence of instructions that is executed repeatedly while or until some condition is met or satisfied. Looping is fundamental to all scripting languages. *JavaScript*, like most scripting languages, has numerous looping constructs.

3.1 The For Loop

This is the most commonly used loop and is applicable when the number of iterations of the loop is known ahead of time. The syntax is:

```
for (start; condition; continuation expression )
{
        Code that is to be repeated a certain number of times
}
```

The *start* is where the variable that controls the loop, the **looping variable** (or **loop variable**), is declared and/or initialized. The *condition* is an expression that establishes the condition(s) for which the loop will continue to run or shall be terminated. At the end of an iteration, the looping variable will be updated (increased or decreased) based on the contents of the *continuation* expression. The current value of the looping variable will now be checked against the *condition* expression; and if still in compliance, the next iteration will be executed, otherwise the loop will terminate.

Consider a loop driven by a looping variable *lpvar* with the *continuation* and *condition expressions* as follows:

```
for (var lpvar = initialvalue; lpvar < terminalvalue; lpvar++ )
{
        Code that is to be repeated until condition statement returns value of false
}
```


In this case *lpvar* is declared and assigned an initial value. It is an integer value. The code will then run for the first iteration. The looping variable will then be incremented to the next integer value. This current value of the looping variable will be checked for compliance with the condition expression. If in compliance, the next iteration will be executed, otherwise the *for* loop then terminates, and the browser continues executing the code on the next line after the loop's closing curly bracket.

It is pertinent to note that the declaration of the looping variable may be implemented before and outside of the looping structure. The increment/decrement of the looping variable does not have to be in steps of unity (1). If necessary, any integer value increment/decrement may be implemented by modifying the increment/decrement operation accordingly (e.g. *k++2* will give an increment of 2, etc.). Also, the loop may be run "backwards", i.e., a higher initial value is assigned, and the *continuation* expression decrements the looping variable down towards the limit set in the *condition* expression. For example:

```
//create your looping variable to control the loop
var lpvar ;

for (lpvar = initialvalue; lpvar >= terminalvalue; lpvar- - )
{
        Code that is to be repeated until condition statement returns value of false
}
```

In the following example, use a *for* loop to create an array of ten elements such that the value of each element is its index value multiplied by 5.

```
//create the array
 var  myArray[0, 0, 0, 0, 0, 0, 0, 0, 0, 0] ;
//declare your looping variable
var i  ;
//loop through the array to calculate and assign array element values
 for(i  = 0;  i = < 9; i++)
{
      mrArray[i] = i * 5 ;}
```


This will result in the array [0, 5, 10, 15, 20, 25, 30, 35, 40, 45].

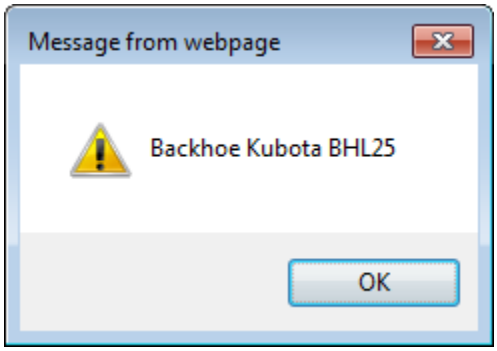
3.2 The For In Loop

The *for in* loop is used to loop through the properties of an object. In some cases the number of iterations may not be known beforehand, which is not an issue because the browser will loop through all the attributes of the object anyway.

Consider an object *Equipment*,

```
var txt = "";
var Equipment = {description:"Backhoe", make:"Kubota", model:"BHL25"};

for (var x in Equipment)
{
    txt = txt + Equipment[x] + " ";
}

alert(txt);
```

So, after looping through the object's attributes and "welding" them on successively, the final result is:



3.3 The While Loop

If the number of iterations needed is not known *a piori*, the *for* loop cannot be used. In that case a *while* loop structure may be used. In the *while* loop, you set the conditions upon which the loop will terminate but will not know ahead of time how many iterations will actually run until termination occurs.

The syntax is as follows:

```
while (condition)
{
        Code to be executed repeatedly while condition holds true
}
```

The termination *condition* is generally some logical expression. The loop variable will be checked against the *condition*. If the *condition* holds true then the first iteration of the code enclosed in the brackets will execute. The loop variable will then be updated to a new current value. The current value of the loop variable will be checked against the *condition* and if it is in compliance, the next iteration proceeds. This procedure will repeat over and over as long as the loop variable meets the condition. If at some point the loop variable does not meet the *condition*, the loop will terminate and the cursor moves to the line after the closing curly bracket.

Consider the following script;

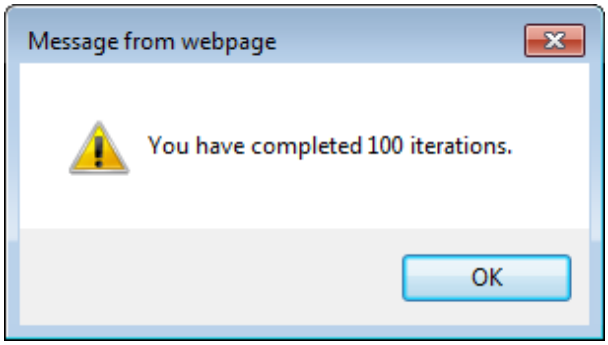
```
//declare and initialize the loop variable
 var i = 0 ;

//loop through values up to 99
while (i < 100)
{
        //write out the value on the web page on a new line each time
        document.write(i +"<br>") ;
```



```
        //go to the next i value
        i++ ;

}

        //alert the user that the task is complete.
        alert("You have completed " + i + " iterations.");
```

This yields (partial screen capture shown):

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31



When using the *while* loop, the termination condition must be chosen carefully and studied closely otherwise the script may fall into an **infinite loop**. An infinite loop is a loop that lacks a functioning exit routine . As a result, the loop cannot stop, and repeats continuously until the operating system senses the issue and terminates the script, or until some event occurs, for instance having the script terminate automatically after a certain duration or number of iterations. Typically, an infinite loop will cause your web browser to crash. Likewise, if you forget to increment (or decrement) the loop variable, the loop cannot end, and your web browser, web page(s), and potentially your operating system, will crash. *While* loops must be used with caution.

3.4 The Do While Loop

The *do while* loop is a variation of the *while* loop. In the *while* loop the condition is checked at the beginning of the iteration. In the *do while* loop, the condition is checked at the end of the iteration, and if the condition fails, the next iteration does not happen and the loop is terminated. The syntax is as follows:

```
do
{
        Code to be executed repeatedly while condition holds true
}
while(condition) ;
```

The *while* loop example we did in the previous section can be set up as a *do while* loop as follows:

```
//declare and initialize the loop variable
 var i = 0 ;

//loop through values up to 99
do
{
        //write out the value on the web page on a new line each time
        document.write(i +"<br>") ;
```



```
        //go to the next i value
        i++ ;

}
while(i < 100) ;

//alert the user that the task is complete.
        alert("You have completed " + i + " iterations.");
```

3.5 Nested Loops

A nested loop is a loop inside of another loop. Loops of all types discussed so far may be embedded (nested) within each other as needed to achieve the desired functionality. All types of loops may be nested within all types of conditional statements (*if* statements) and vice versa, without limit. The same applies for composite conditional statements. It is the responsibility of the web programmer to devise and design the appropriate branching and looping structures to meet the objectives of the project.

3.6 Exiting Loops

In some cases it may be necessary to abruptly or prematurely exit a loop based on the progress of the script. In *JavaScript*, the *break* statement is used to "jump out" of a loop and continue execution from the line of code after the loop, if any. The syntax is:

```
break ;
```

Consider the following script:

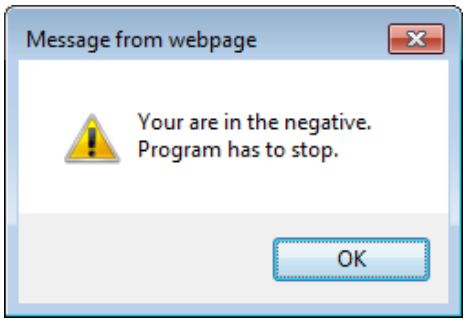
```
<script>

        //declare loop variable and a variable to hold the result of a calculation
        var x=0.1,i=0;
```



```
//loop from 20 down to zero and perform the calculation each time
for (i=20;i>0;i- -)
  {

//the calculation
   x = (i/2) - 4;

//but we don't want negative values in our results
//if that happens stop the script
//so we shall nest an if statement in our loop
            if (x < 0)
             {

             //if we get a negative result, display the values
             //of i and x

             document.write("i is " + i + "<br>");
             document.write("x is " + x);

             //alert the user that the program is about to shut off
             alert("Your are in the negative. \nProgram has to stop.");

             //terminate the loop with a break call
             break;
             }

  }

</script>
```

The following shows when a negative result (x) was encountered, and the value of the loop variable (i) that caused it.

i is 7
x is -0.5



A code block can be labeled. A code block can be prematurely exited by calling the *break* statement and the name of the label. The previous example may be modified as follows.

```
<script>

        //declare loop variable and a variable to hold the result of a calculation
        var x=0.1,i=0;

stopcondition: //this is the label for the for loop with if statement code block
                //the entire block of labeled code must be wrapped in curly brackets

{

        //loop from 20 down to zero and perform the calculation
        for (i=20;i>0;i- -)
          {

        //the calculation
          x = (i/2) - 4;
```



```
//but we don't want negative values for the calculation result
//if that happens stop the script
//so we shall nest an if statement in our loop
        if (x < 0)
         {

         //if we get a negative result, display the values
         //of i and x
        //post on web page that loop was stopped
         document.write("Premature termination !"<br>");
         document.write("i is " + i + "<br>");
         document.write("x is " + x);

         //terminate the code block with break call with label
         break stopcondition;
         }//close of if-statement

    }//close of for-loop

 //alert the user that the program has run successfully to completion
//this will happen only if the termination condition in the code block
//is never met. We can expect this to never happen as we already know
//that the calculation will yield a result that will trigger the termination condition
 alert("Successful Completion!");


}//close of labeled code block

</script>
```

The script yields the following:


```
Premature termination !
i is 7
x is -0.5
```

Note that in this case the alert for successful completion never pops up because the condition for termination occurs in the labeled code block, and the labeled code block is then called to terminate with the *break* statement once the condition for termination (in the *if* statement) is met. Once "kicked out", the cursor resumes execution on the line after the line with the closing curly bracket of the labelled code block.

A variation of the *break* statement is the *continue* statement. The *continue* statement can be used within a loop to "jump over" one iteration based on some condition. The *continue* statement may be used in conjunction with a label, but the entire construct (label, code block, and *continue* statement with label) must all be within the structure of the loop.

4. OBJECTS

4.1 Objects in *JavaScript*

An object is data that has properties and methods stored with it. The properties are values that describe the object whereas the methods are actions associated with the object. A *JavaScript* object is essentially a collection of properties with values assigned them that altogether describe some data.

Consider a house as an object. The properties of the house would include, for example, stories, structure, bedrooms, bathrooms, roof, flooring, exterior, etc. The methods would include, for example, paint, clean, fix, cut grass, renovate, etc.

There are a number of ways to create an object in *JavaScript*. One way is to use the keyword *new* to create the object and assign it to a variable followed by a list of property-value assignments. For our house example, for instance,

```
var xyHouse = new Object( ) ;

        xyHouse.storeys = 2 ;
        xyHouse.structure = “concrete block” ;
        xyHouse.bedrooms = 4 ;
        xyHouse.bathrooms= 3.5 ;
        xyHouse.roof =”sheet metal”4 ;
        xyHouse.flooring = “ceramic tile” ;
        xyHouse.exterior = “brick” ;
```

Object methods are functions we can develop that will manipulate some property or the other of our object. For example *xyHouse.Paint( )*, or *xyHouse.ChangeRoof( )*, etc.

An object property may be accessed as follows,

```
 z = xyHouse.exterior ;
document.write(z) ;
```


which will return “brick”.

Another way of creating an object is by using the **object literal**, as follows,

```
var xyHouse {storeys: 2, structure: ”concrete block”, bedrooms:5,  bathrooms: 3.5}
```

or

```
var xyHouse {
            storeys: 2,
            structure: ”concrete block”,
            bedrooms:5,
            bathrooms: 3.5
            }
```

Another way of creating an object is by using an **object constructer function** (or **constructor**), as follows,

```
function xyHouse(a, b, c, d)
{
this.storeys = a;
this.structure = b;
this.bedrooms = c;
this.bathrooms = d;
}
```

The *this* keyword sets the properties of the constructor. The advantage of using the constructer is that the object (*xyHouse*) now becomes a custom object type. We can now quickly create new instances of the *xyHouse* object. For example,


```
var dadsHouse = new xyHouse( 1, "wood frame", 3, 2.5) ;
var johnnyHouse = new xyHouse(2, "steel frame", 5, 4.5) ;
```

JavaScript has many built- in objects, each with its constructor, built-in properties and methods. In fact, all of the data types we have learned thus far can in one way or the other be modelled as objects. In the remainder of this chapter we shall review a selection of *JavaScript* built-in objects. A comprehensive review of *JavaScript* objects can be found elsewhere. (w3schools.com, 2014a)

4.2 Date

The date object is used for the storage, retrieval and manipulation of dates and times. A date may be created with the date constructor in any of the following ways.

i. *dateobjectvariable = new Date( ) ;*

This will create the current date and assign it to the variable. So if today is May 2, 2014, 4:45 pm on the east coast of the United States,

```
var dteVar  = new dDate( ) ;
document.write(dteVar) ;
```

Which returns "Fri May 2 16:45:17 EDT 2014" (the date and time this code was executed).

ii. A date may be created by passing the number of milliseconds in Universal Coordinated Time (UTC) that have passed since midnight January 1, 1970, to the date constructor. Noting that a day consists of 86,400,000 milliseconds,

```
var dteVar  = new Date(1135296000000 ) ;

document.write(dteVar) ;
```


This returns “Thu Dec 22 19:00:00 EDT 2005”.

iii. Passing a date string to the date constructor will create a date object

```
var dteVar  = new Date( “May 2, 2014 14:20:00”) ;

document.write(dteVar) ;
```

This returns “Friday May 2 14:20:00 EDT 2014”.

Note that as the time zone was not specified, the local time zone is applied, in this case EDT for the east coast of the United States.

iv. Passing any number of the date parameters to the date constructor will create the date object.

```
new Date(year , month, day, hours, minutes, seconds, milliseconds);
```

where January is month zero, and December is month 11. For example,

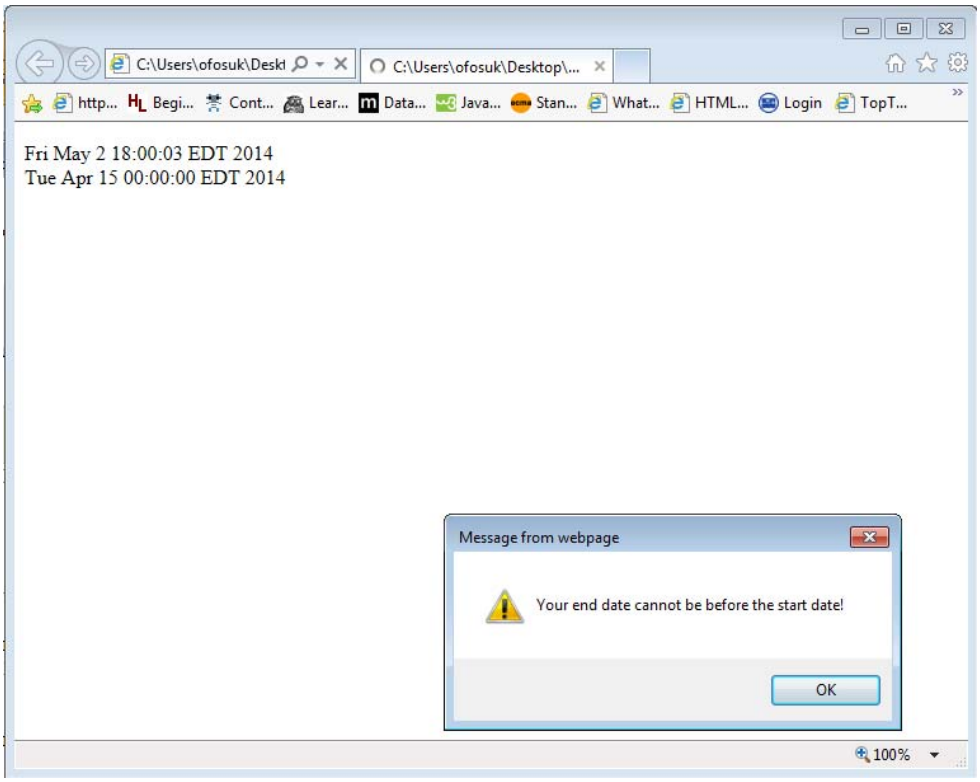
```
var dteVar  = new Date( 79, 2, 18, 10,20,00) ;

document.write(dteVar) ;
```

which yields “Sun Mar 18 10:20:00 EDT 1979”.

Dates can be compared. For example the following code checks if the user has inconsistent start and end dates for a project.


```
<script>

var startdate = new Date( );
document.write(startdate + "<br>");

var enddate = new Date("April 15, 2014");
document.write(enddate + "<br>");

if (enddate < startdate)
{
        alert("Your end date cannot be before the start date!");
}

</script>
```

Resulting in,



Some of the commonly used date methods are presented in Table 4.1

Table 3: Date methods

Method	Description	Example	Result
getDay( )	Extracts weekday of a date and converts it to a number	*var dte = "May 2, 2014"* *document.write(dte.getDay( ))*	5
toUTCString( )	Converts a date string to the UTC date format	*var dte =* "Fri, 2 May 2014 23:44:02 EDT" *document.write(toUTCString( ))*	Sat, 3 May 2014 03:44:02 UTC
getTime( )	Returns the number of milliseconds elapsed since Midnight January 1, 1970	*var dte =* "Fri, 2 May 2014 23:46:02 EDT"	1399088802978
getFullYear( )	Returns the full year of a date string	*var dte = "May 2, 2014"* *document.write(dte.getFullYear( ))*	2014

4.3 Strings

A string is a series of characters. Strings are used to store and manipulate text. A string is assigned to a variable by placing it within double or single quotation marks. For example,

```
var strFirstName = "Cameron" ;

var strLastName = 'Milner' ;
```

Each character has an address or position within the string called its **index**. The first character is of index zero [0], the next character index [1], and so on and so forth. For example,

Each character in the string can be accessed by calling it index. For example,


```
var strLetter = strFirstName[4] ;

document.write(strLetter);
```

Which returns a value of "r".

To incorporate quotation marks in the string, the "\" is put in front of the quotes sign, as follows:

```
var strGreeting = "Say \"HELLO\" to all of your customers." ;
document.write(strGreeting);
```

Which returns "Say "HELLO" to all of your customers. Alternately, you may use one style of quotes (single or double quotes) within another style to obtain the same effect. For example,

```
var strGreeting = 'Say "HELLO" to all your customers.' ;
document.write(strGreeting);
```

Which returns "Say "HELLO" to all of your customers."

The *length* property of the string gives the number of characters in the string. For example,

```
var txtName = "Eyjafjallajökull" ;
document.write(txtName.length) ;
```

Which yields a value of 16.

Case Conversion

The built-in functions *toUpperCase( )* and *toLowerCase( )* will convert a string to upper case or lower case respectively. For example,

```
var strNameEntry = 'john henry' ;
var UCaseName = strNameEntry .toUpperCase() ;
document.write(strUCaseName);
```

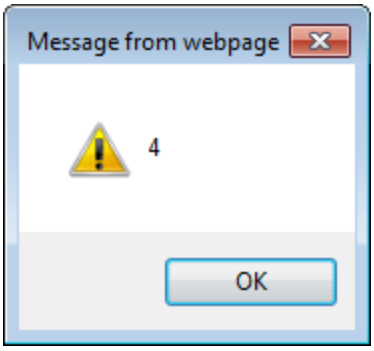
This returns "JOHN HENRY".

Sub Strings

The *indexOf( )* method will return the starting position (index) of a sub string within a larger string (keep in mind that indices in *JavaScript* start from zero). The *lastIndexOf( )* method will locate the sub string's starting position within a larger string, but will start the search from the end of the string. In either case, if the sub string is not found, the respective function will return a value of -1. For example,

```
var strSalutation = 'Transylvania' ;
var indexLeft = strSalutation.indexOf("sylvan") ;
var indexRight = strSalutation.lastIndexOf("sylvan") ;

alert(indexLeft) ;
alert(indexRight);
```

This yields:

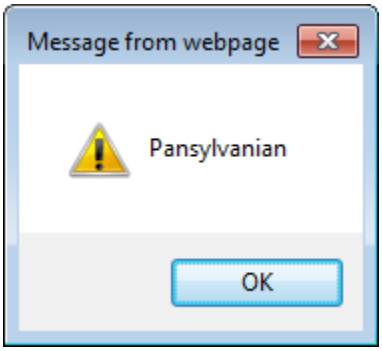


for either case.

The *match( )* method will search for the matching content in a string. The *replace( )* method will search for a specific content in a string and replace it with specified string. For example,

```
var strSalutation = 'Transylvanian' ;
var indexNew = strSalutation.replace("Tran", "Pan") ;

alert(indexNew);
```

Which yields,



Special Characters

The back slash (\), in conjunction with other characters, creates special characters in *JavaScript*. Table 4.2 presents some special characters in *JavaScript*.

Table 4: Special characters

Special Character	Output/ Keyboard Equivalent
\b	backspace
\n	new line
\r	carriage return
\t	tab
\\	backslash
\'	single quote
\"	double quote

Comprehensive lists of special characters as well as other properties and methods for strings can be found at various sources. (ECMA International, 2011), (Microsoft, 2014b).

Objects

Strings can also be created as objects. For example, the following creates a custom string type.

```
var ks = new String("Hector") ;
```

4.4 Arrays

An array is a variable that holds multiple values. The individual values are called the **elements** of the array. Each element is identified by its unique address in the array called its **index**. In *JavaScript*, the first element is of index zero [0]. The elements of an array may be of different data types. In large and complex applications, arrays drastically reduce the number of variables needed. An array is created as follows:


```
var NameOfArray = [firstelement, secondelement,   , nthelement ] ;
```

or,

```
var NameOfArray = [
                   firstelement,
                   secondelement,
                   :
                   :
                    nthelement ] ;
```

An element of an array is accessed by referring to its index number. For example,

```
ProjectTeam[0] = "Paul Andrews, P.E." ;
ProjectTeam[1] = "Marta Gomes, P.E." ;
ProjectTeam[2] = "Donald Keino" ;
ProjectTeam[3] = "Paula Generale" ;
```

Alternately, an array may be created as an object using a constructor, as follows,

```
var NameOfArray = new Array( ) ;

NameOfArray[0] = firstelement ;
NameOfArray[1] = secondelement ;
NameOfArray[2] = thirdelement ;
   :
   :
   :
NameOfArray[n] = nthelement ;
```


or,

```
var NameOfArray = new Array(firstelement, secondelement,   , nthelement ) ;
```

For example,

```
var ProjectTeam = new Array( ) ;
        ProjectTeam[0] = "Project Manager" ;
        ProjectTeam[1] = "Design Engineer" ;
        ProjectTeam[2] = "Engineering Assistant" ;
        ProjectTeam[3] = "Drafter" ;
```

This yields the array:

$$\begin{bmatrix} Project\ Manager \\ Design\ Engineer \\ Engineering\ Assistant \\ Drafter \end{bmatrix}$$

Array objects have predefined properties and methods. A comprehensive presentation of the built-in properties and methods of arrays in *JavaScript* can be found at other sources. (Mozilla, 2014). In the remainder of this section we shall review a selection of built-in *JavaScript* array properties and methods.

Length

The length of an array is the number of elements it has. From our project team array above,

```
var numStaff = ProjectTeam.length ;
document.write(numStaff) ;
```


This will return a value of 4.

Sort

The elements of an array can be rearranged in order, alphabetically or numerically using a *sort* method. For example,

```
ProjectTeam.sort( );
```

will rearrange the elements alphabetically as *["Design Engineer", "Drafter", "Engineering Assistant", "Project Manager"]*

Consider the array with elements as follows,

```
var arrWage = [24.5, 16.98, 24.98,19.35] ;
```

To arrange the elements in ascending order, use:

```
arrWage.sort( ) ;
```

or,

```
arrWage.sort(function(a,b)[return a-b]) ;
```

To arrange the elements in descending order, use:

```
arrWage.sort(function(a,b)[return b-a]) ;
```


Concatenation

This is the method of "welding" two arrays together. For example,

```
var arCivilProjects = ["Pinewood Subdivision", "KO Gas Station", "Placido Plaza"] ;
var arRoadProjects = ["Dunwoody Parkway", "Nejame Avenue"] ;

var arAllProjects = arCivilProjects.concat(arRoadProjects) ;
```

The array *arAllProjects* will have the elements *["Pinewood Subdivision", "KO Gas Station", "Placido Plaza", "Dunwoody Parkway", "Nejame Avenue"]*

Convert to string

The *toString( )* method converts an array to a text string. For example,

```
arRoadProjects.toString( ) ;
```

This yields the text string *"Dunwoody Parkway, Nejame Avenue"*.

Shift( )

The *shift( )* method will remove the first element from the array. For example,

```
arCivilProjects.shift( );
```

This yields the array *["KO Gas Station", "Placido Plaza"]*

4.5 Practicum #4

In this practicum we shall add further functionality to our yellow time calculator. We shall add a "Batch" calculation button. On clicking the batch calculation button, the yellow time will be calculated for speed scenarios starting at 15 mph in increments of 1 mph up to 80 mph.

For each speed, a yellow time will be calculated based on the other inputs. A transcript in the form of a table will be generated at the end of the procedure and displayed on the web page.

Solution

Save a copy of your Practicum #3 file and rename it Practicum #4.

Open Practicum #4 with your favorite text editor. In this practicum, the text editor *Notepad++* shall be used. This is one of many downloadable, free, open source software that support code writing and editing in several programming, as well as scripting languages. (Ho, 2011).

After the entry validation checks, add a loop that will calculate the yellow time from *speed* =14 through *speed* = 80 in increments of 1. A partial screenshot of the loop is as follows.

```
//we shall use a for loop to perform the iterative calculations
//for each speed value, holding the other inputs constant
for(var speed = 14; speed <=80; speed++)
{
//calculate the yellow time using the ITE formula
yellowtime = reacTime + 1.47*speed/(2*decel+32.2*gradient/100) ;
```

Each time a calculation is performed we want to compile the results somewhere such that at the end of the procedure we shall display the compiled results. Create a variable to store this information. We shall use a string for this purpose. Declare it before your *for* loop.

```
//set up a string to compile our results
//(note that an array can also be used)
//we shall also use it to display the results later
var kspeed = " ";

//we shall use a for loop to perform the iterative calculations
//for each speed value, holding the other inputs constant
for(var speed = 14; speed <=80; speed++)
{
//calculate the yellow time using the ITE formula
yellowtime = reacTime + 1.47*speed/(2*decel+32.2*gradient/100) ;

//assign calculated value to the string box to compile it
```


We actually want to compile results from speed of 15 mph. We are using the start value of "14" as a dummy value to enable us to set up a header for our output table. So if the speed is 14 mph, set up and append a header to the string variable compiling our results, otherwise display the speed and the calculated yellow time. We can implement this with an *if* statement as follows:

```
//set up a string to compile our results
//(note that an array can also be used)
//we shall also use it to display the results later
var kspeed = " ";

//we shall use a for loop to perform the iterative calculations
//for each speed value, holding the other inputs constant
for(var speed = 14; speed <=80; speed++)
{
//calculate the yellow time using the ITE formula
yellowtime = reacTime + 1.47*speed/(2*decel+32.2*gradient/100) ;

//assign calculated value to the string  to compile it
//and later to display it
//if speed is 14 mph create a header row
//we want to start compiling results from 15mph
if (speed==14)
{
//this is the header row
kspeed = kspeed + "speed (mph)" + " " + "yellow time (seconds)" + "<br>";
}
else
{
//the actual results
//note the use of the special charachters to tab and space out the results
//to get a good looking table
kspeed = kspeed + " " + speed + "     " + yellowtime + "<br>";
}


}//end of for loop
```

Once all calculations have executed and been compiled in our output string, we want to display the results. We can use *document.write*, etc. We may also use the *innerHTML* property of an element. In this case we shall create a paragraph towards the bottom of our web page and give it

a *name* (or *id*) attribute. We shall use the *innerHTML* property to send our output text string to that paragraph to be displayed on the web page.

Scroll towards the bottom of your web page source code and add a paragraph called *batchrun*.

```
                          
                        

                <button type = "button" name = "calcAllredTime" onclick = "fnAllredCalc()">Calculate</button>

                      

                <button type = "reset" name = "resetAllredTime">
                                  Reset  </button>

            </p>

            <br><br><!--this is where we shall post the results of batch runs-->
             <p id="batchrun"> </p>

            </form>


        </td>

        <td width = "20%">

        </td>

    </tr>


</table>

        <p><br><b><center>&copy; March 2014. All Rights Reserved</center></b></p>
```


Back in our batch function procedure, after the *for* loop, we shall add code to assign our output string to the element (the paragraph) called *batchrun*, by accessing its *innerHTML* property as follows:

```
        //this is the header row
        kspeed = kspeed + "speed (mph)" + " " + "yellow time (seconds)" + "<br>";
        }
        else
        {
        //the actual results
        //note the use of the special charachters to tab and space out the results
        //to get a good looking table
        kspeed = kspeed + " " + speed + "     " + yellowtime + "<br>";
        }


        }//end of for loop

        //once compiled, lets display the string.
        //at the paragraph named (id) "batchrun"
        document.getElementById("batchrun").innerHTML = kspeed ;
        //once complete, tell user
        alert("Scroll towards the bottom of the page to see your results.") ;
    }
    else
    {//if you press Cancel nothing will happen

    }


    return ;
}
```

An alert message was added to inform the user that the calculations have run to completion and the user may scroll down the web page to review the results.

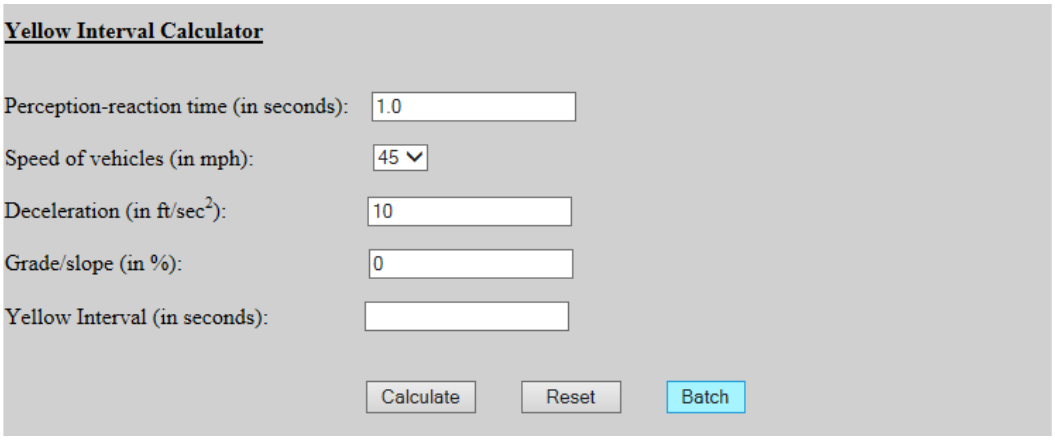
Finally, we shall embed our entire *batch* code within a confirm box structure so that the user can be cautioned on what is about to happen and choose whether to continue with the *batch* execution or cancel out of it. A partial screenshot is as follows (the *else* condition and closing curly brackets towards the bottom are not shown):

```
  </title>

<script type="text/javascript">
//<script language="JavaScript">


 function fnBatchYellow()
{
    //use confirm box to alert user that they are about to do a batch run fo
    //yellow time over several speed values

    // declare variable for the confirm box
    var  xconf = confirm("You are about to run the Batch procedure. \nClick OK to continue.")  ;

    //if user clicks on Ok to continue with the batch calculation
    if (xconf == true)
    {

        //declare variables needed for the calculation
        var reacTime, decel, gradient, yellowtime ;

        /*now assign the controls' values to the variables, identifying
        each control by its unique id */
        reacTime = document.getElementById("txtReacTime").value;

        decel = document.getElementById("txtDecel").value;
        gradient = document.getElementById("txtGrade").value;

        //check that the entry for reaction time is a valid numeric value
```

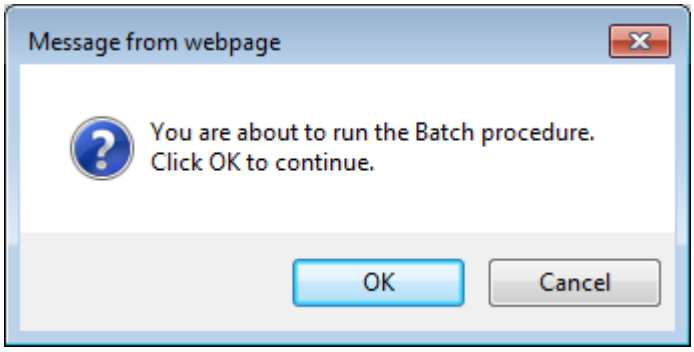
Review your codes and check for common errors, omissions, typos, etc. For example, confirm that all *if* statements, loops, etc., have all their corresponding opening and closing curly brackets in place. Check quotation marks and confirm that all pairs are correctly in place. Check for missing "**;**" wherever they are required.

Save your file.

Test your file.
Open your file in your web browser.
Enter reaction time, deceleration, and grade inputs.
Click on *Batch*.

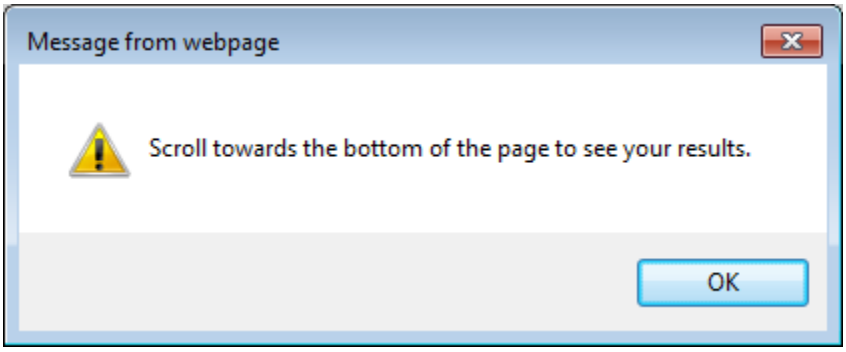


The confirm box pops up to confirm whether the user would like to continue or cancel.
Press *OK* to continue.

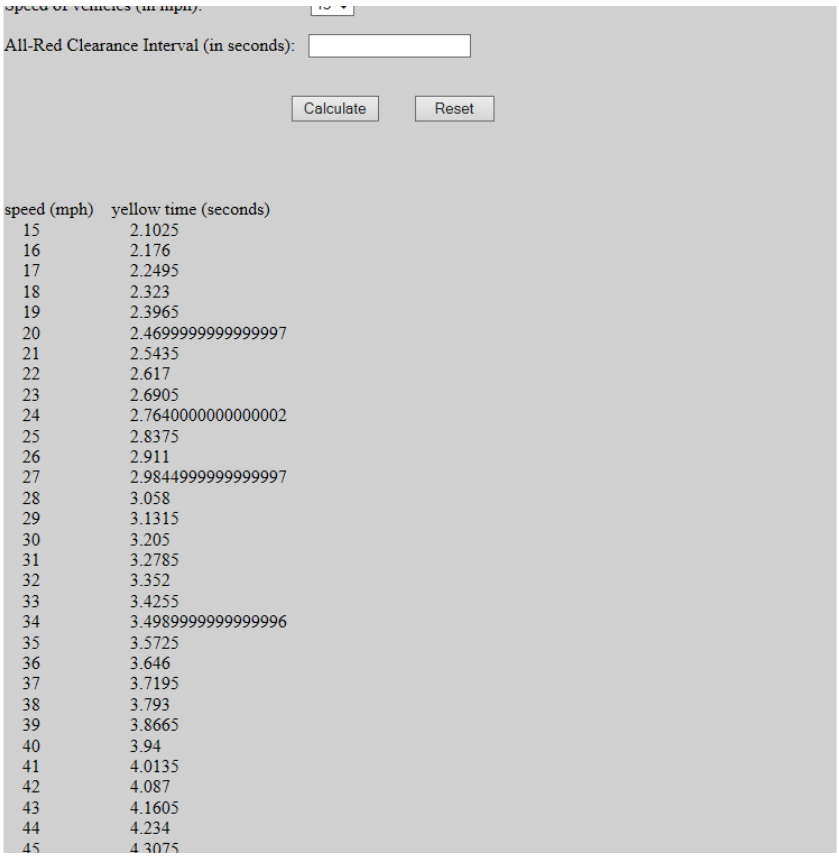


The inputs will be checked and validated, and then the iterative calculations will proceed.

The user is alerted that the procedure has run to completion.



Dismiss the alert to review the results which are displayed in the paragraph towards the bottom of the web page. A partial screenshot is as follows:



The test is a success.
You have created a fully functional interactive web page that gives the user the opportunity to play "traffic engineer" and conduct sample yellow time calculations regarding the Red Light Camera program.

For further independent study, consider the following alternatives/ improvements:

- Use an array to compile and display the batch results.
- Apply *break* statements with labels to implement the validations, etc.
- Implement a consistent number formatting for all calculated results.
- Center your results output on the web page.
- Make the headers bold, underlined, and distinct from the output data.
- Add code that will also save the output of the batch run to an external file such as a text file, pdf, *Word* document, *Excel* spreadsheet, etc., etc.

5. MANIPULATING THE BROWSER

5.1 The Window Object

The Browser Object Model (BOM) enables *JavaScript* to communicate directly with the web browser. The *window* object represents the web browser window.

Examples of *window* properties include the window height and the window width as follows:

```
var w = window.innerHeight //the inner height of the browser window is saved to a variable
 var h = window.innerWidth //the inner width of the browser is saved to a variable
```

Examples of windows methods include:

```
window.open( ) //opens a new browser window
window.close( ) // closes the current window
window.moveTo( ) //moves the current window
window.resizeTo( ) //will resize the current window
```

5.2 The Screen Object

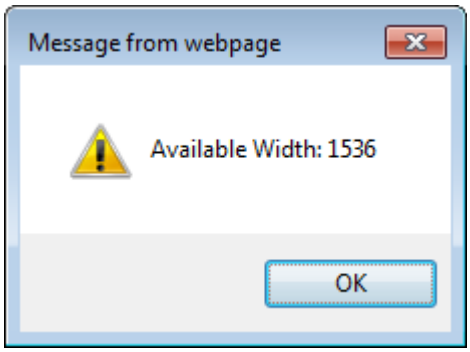
The *screen* object holds information about the user's screen. Some properties include:

```
screen.availWidth //the available width of the user's screen
screen.availHeight //the available height of the user's screen
screen.pixelDepth //the color resolution of your screen
```

For example:


```
<script>

    alert("Available Width: " + screen.availWidth);

</script>
```

This yields the following on the author's laptop (your result may vary):



You are encouraged to test this on your equipment and see what value you obtain.

5.3 The Location Object

The *location* object is used to extract the URL of the current web page. Some example properties include:

```
location.href          //will return the url of the current web page
location.hostname      //will return the domain name of the web host
location.pathname      // will return the path and filename of the current web page
location.port returns  //the port of the web host (80 or 443)
location.protocol      //will return the web protocol used (http:// or https://)
```


Test the following code on your equipment.

```
<script>
        alert("Current URL: " + location.href);
</script>
```

An example of a location method is:

```
location.assign( )       //this method will load a new document
```

For example:

```
location.assign("http://www.suncam.com")  //note that the full url is required
```

5.4 The History Object

The *history* object contains your browsers history. Due to privacy considerations *JavaScript* has limitations on how it accesses this object. Examples of methods include:

```
history.back( ) //this loads .the previous URL in the history list stored in the history object.
                //this is the same as clicking on the back button of your browser

history.forward( )   //this is the same as clicking on the forward button of your browser
```


5.5 Timing of Events

It is possible to execute certain codes at specific time intervals. This is referred to as **timing events**.

The ***setInterval( )*** method will wait for the specified number of milliseconds and then run a specified function. It will then wait for the specified time interval and run the function again. This cycle will repeat continuously. **The *clearInterval( )*** method is used to terminate further execution of the function specified by your *setInterval( )* method. The following example will be used to demonstrate the *setInterval( )* and *clearInterval( )* methods.

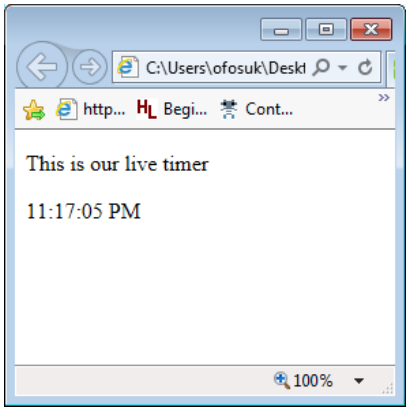
Example: Create a "live" clock for a website. The clock starts ticking once the web page opens. The clock shall display the local time in hours, minutes and seconds. The code to set up the clock is as follows:

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

<p>This is our live timer </p><br><br>

<!-->paragraph where the time will be displayed by accessing
the innerHTML of the paragraph, using its id attribute <-->
<p id = "display"></p>

<script>

        // the setInterval method syntax is as follows

        //setInterval(function( ){nameoffunction ( )}, repeatinterval) ;
        //you may write the code for the function within the curly brackets or as we shall do
        //in this example, quote the function name and write the code for it elsewhere
        //in the script
```



```
        //in this case we repeat the function every 1000 milliseconds (which is 1 second)

        //we shall save the output to a variable which we shall
        //later call to display its contents on the web page
        var myTimer=setInterval(function(   ){myClock( )},1000) ;

        //now we set up the function myClock which we shall repeat at the stated interval
        function myClock( )
        {

        //first give us the current date and save it to a variable
        var ctime=new Date( ) ;

        //convert the date given to local time as a regular string using the
        //built-in function toLocaleTimeString( )
        var localTime=ctime.toLocaleTimeString( ) ;

        //now display the time on the web page in the
        //paragraph called "display" towards  the top
        document.getElementById("display").innerHTML = localTime ;


        }//end of function

</script>



</body>

</html>
```


Test the code in your browser.
The function to call the time, convert it to local time, and display it on the web page, is re-executed every 1000 milliseconds. The user sees a "live, ticking" clock.



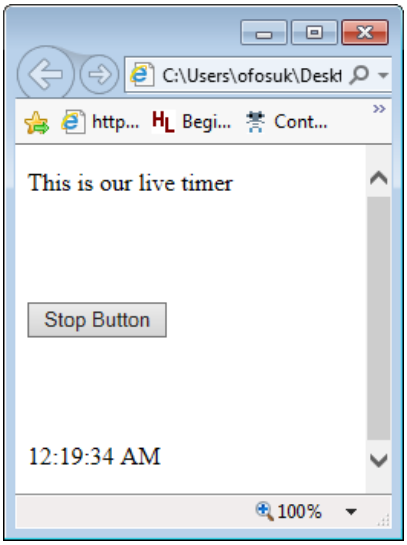
We shall now add a button which when clicked on stops the clock.

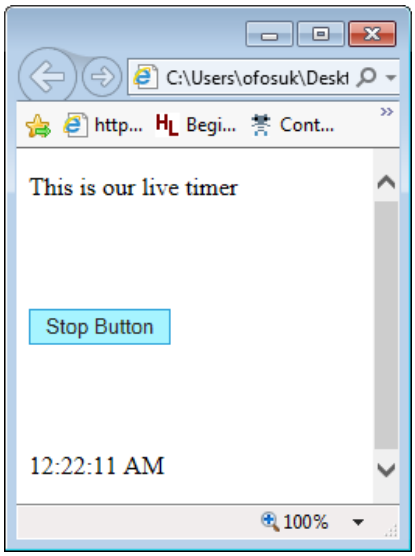
```
<!DOCTYPE html>
<html>
<body>

<p>This is our live timer </p><br><br>

<p><!—add a stop button to the web page><-->
<button onclick="stopTimer()">Stop Button</button>
</p>

<br>
<br>

<p id = "display"></p>
```



```
<script>

var myTimer=setInterval(function(){myClock( )},1000) ;

function myClock()
{

var ctime=new Date( );

var localTime=ctime.toLocaleTimeString();

document.getElementById("display").innerHTML = localTime;

}

//add function that executes when the stop button is pushed

function stopTimer()
{

        //use clearInterval method to stop the timer, we use syntax
        //clearInterval(name of timer variable);

        clearInterval(myTimer);

}

</script>

</body>
</html>
```


Test your code.
Load the web page. The live clock displays the new time every second.
Push the button to stop the clock.





The test is a success. You have successfully created an interactive clock for a website.

Another *JavaScript* timing event is the ***setTimeout( )*** method. The *setTimeout( )* method will wait for a specified number of milliseconds and then execute the specified function. The function will run once and will not repeat as we saw with the *setInterval( )* method. The syntax is,

```
variable = setInterval(function( ){nameoffunction( )},milliseconds) ;
```

The *setTimeout( )* can be stopped by calling the ***clearTimeout( )*** method, and passing to it the variable that was assigned to the *setTimeout( )* method. The syntax is as follows,

```
clearTimeout(variable) ;
```

5.6 Cookies

A cookie is a text file that contains some data, put on your computer by a web page when you visit it for the first time. The cookie enables the web page to "remember" who you are. When you log back onto that web page, the cookie, stored in your web browser, tells that web page who you are, how many times you have visited that web page, your log in rights and status, the information you look at often, etc., etc. Cookies can store your user name(s), preferences, settings, etc., for specific web sites. While cookies may be abused in terms of gathering personal data of the users, they are not by themselves malicious.

A cookie has a *name* parameter which is assigned a value. A cookie has an *expiry* parameter which is the expiry date of the cookie. The expiry date should be given in the *UTC (Greenwich) time* format. Once the expiry date passes the cookie is deleted permanently from your computer. If the *expiry* parameter is not set, the cookie will be deleted when you close your web browser.

A cookie has *domain* and *path* parameters. The domain is the website (or sever, etc.) the cookie originated from. A cookie set by a particular domain can only be subsequently read by and from that domain and its sub domains. For example, a cookie set on my computer by *microsoft.com* cannot be subsequently read by *yahoo.com* and so on. If the domain is not explicitly set, then the current domain (web site) becomes the domain by default.

The *path* parameter is used to set a specific directory of the domain from which the cookie can be read. So for example, if the path parameter is set to *espn.com/football*, it cannot be read by a web page "sitting" in *espn.com/hockey*. It is common practice to set the path to "/", which means that the cookie can be read from throughout the entire domain of the website (or server) from which it originated. Also, you cannot make a domain you are currently not in your cookie domain. For example, while in the *espn.com* domain (in other words with an *espn.com* web page currently active), I can make *espn.com* or any of its sub directories my cookie domain, but I cannot make say *discoverychannel.com* domain my cookie domain.

The *JavaScript* property *document.cookie* can create, read, and delete cookies. For example, we can create a cookie called *mytestcookie* which expires at 8:30 pm on August 1, 2014, and can be read from the current domain and all of its sub domains, as follows:

```
document.cookie =
"cookiename = mytestcookie ; expires = Mon, 1 Aug 2014 20:30:00 UTC; path = /" ;
```

Note the usage of the quotation marks and the semicolon separator for the parameters.

The contents of a cookie can be read as follows:

```
var x = document.cookie ;
```

The result will be a string listing the *name-value* pairs only of all the cookies from the current domain that have been stored in your web browser. In the above example, the result will be:

```
cookiename=mytestcookie ;
```

A cookie can be changed by re-creating it with the updated information. The old cookie will be overwritten. For example, to change the expiry date of our cookie to October 31, 2014, we simply re-create it and modify the *expiry* parameter as follows;

document.cookie =
"cookiename = mytestcookie ; expires = Wed, 31 Oct 2014 20:30:00 UTC; path = /" ;

To delete a cookie, simply change the date value to a date that has already passed. For example,

document.cookie =
"cookiename = mytestcookie ; expires = Wed, 31 Oct 2010 20:30:00 UTC; path = /" ;

Numerous websites run scripts that use a prompt popup to ask the user for their name which is used to set a cookie. On revisiting the web page (or refreshing the web page) another script checks if the cookie exists. If the cookie exists the user may recieve an alert box with their name on it welcoming them back to the website. If the cookie does not exist, the prompt popup appears and asks the user to enter their name or sign up to that website. (quirksmode.org, 2014;w3schools.com, 2014)

6. ERROR HANDLING

6.1 Errors

It is common, even among seasoned programmers, that due to some error(s) in the code, the script may not work as expected, or it may run partially and prematurely terminate, or not run at all. Identifying errors and addressing them is called **debugging**. In scripting, error handling refers to techniques and practices used to test scripts and isolate errors. When an error occurs, the browser will stop executing the script and **throw** an error message.

6.2 Types of Errors

Errors have several causes. **Syntax errors** are the result of misspelled or omitted keywords, typos, incomplete branching or looping structures, inadmissible use of mathematical operators and functions, among others.

A **run-time error** occurs when some value is processed or some resource is accessed in a manner that is inadmissible to the scripting language. This will cause the script execution to terminate. For example, dividing some value or variable by zero will cause an error as the value is mathematically indeterminate.

Logic errors, commonly called **bugs**, occur when the script runs "normally" but produces unexpected or undesirable results. In other words, upon review, the programmer knows that the results are incorrect, but from the point of view of the browser, the script is "fine". Logic errors also exist if the script behaves erratically; for example, results are displayed to say a text box that was not the intent of the programmer. Due to the fact that the script will be running "normally", there will be no error messages thrown at the user. This makes logic errors difficult to identify. The programmer must have some domain knowledge of the underlying theories and mathematical models being implemented in the script. The script must be tested repeatedly and the results thoroughly scrutinized, and verified and validated.

Some common causes of logic errors include:

- omission of relevant code
- incorrect sequence of instructions
- calling the wrong variables or functions

- incorrect choice of branching and looping structures
- incorrect variables and/ or logic in conditional statements
- incorrect (loop) variables in loops
- incorrect referencing to array indices

6.3 Handling Errors

During script execution, if an error occurs while a script is being executed, the applicable built-in error popup box will be thrown at the user. Ordinarily, a non-expert end-user will not know what the error message means or what steps should be taken to address it. As a result, it is considered unacceptable scripting practice for built-in error messages to be thrown at an end user. It is therefore the programmer's responsibility to anticipate potential errors that may occur and add code that will address them in such a manner that the built-in error messages do not open to an end user. This is the basis of error handling. Errors that are anticipated and addressed such that the relevant built-in error message box does not appear to the user, are referred to as **handled errors**, otherwise they are referred to as **unhandled errors**.

6.4 Try and Catch Statements

The *JavaScript try* statement defines a block of code to be tested while it is being executed. The *catch* statement defines a block of code to be executed if the *try* block picks up an error. The *try* and *catch* statements are used together as a pair as follows:

```
try
  {
        //Block of code to be tested for errors
  }
catch(err)
  {
        //Run this block of code to handle errors
  }
```

where *err* is a variable representing the (built-in) error object (the popup error box) that will be thrown.

Properties of the error object include:

- *description* property: This is the descriptive string on the error popup.
- *message* property: This is a string that contains an error message associated with the error.
- *name* property: This is the name of the error. It describes the type of error that has occurred.
- *number* property: This is an "official reference number" of the error message that has been thrown.

The error object also has methods. For example, the *toString* method will return a string representation of the error popup.

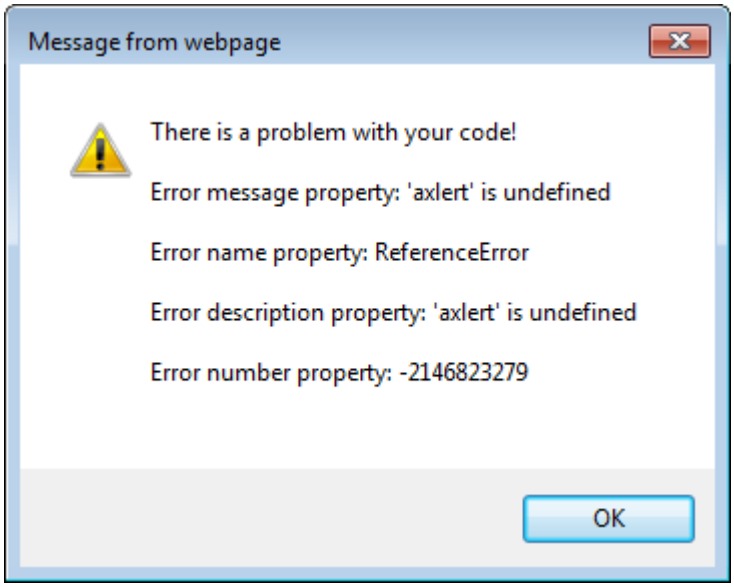
Consider the following code example of a *try* and *catch* pair. The code within the *try* block has intentionally been typed incorrectly, *axlert* instead of *alert*.

```
<script>

//create text string to hold error information that will be displayed on the error popup
var txtVar="";

//insert the code to be tested in a try block
try
 {
        axlert("Welcome to my website!");
 }

catch(err)
 {
   //if an error occurs, extract the following properties of the error object
   //and append to our text variable for subsequent display
        txtVar="There is a problem with your code!\n\n";
        txtVar+="Error message property: " + err.message + "\n\n";
        txtVar+="Error name property: " + err.name + "\n\n";
        txtVar+="Error description property: " + err.description + "\n\n";
        txtVar+="Error number property: " + err.number + "\n\n";
```



```
    //display the text string with the captured
    //error object properties on an alert box
        alert(txtVar);
}

</script>
```

Our "custom built" error message/ alert box that will be thrown is as follows:



Further information on error object methods as well as error object properties and their values can be found at other sources. (Microsoft, 2014a)

6.5 Throw Statement

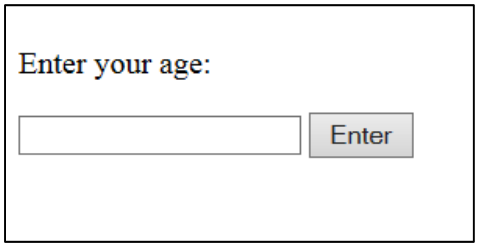
The throw statement enables you to create and throw a custom error. This is described as to **throw an exception**. Throw statements used in conjunction with *try* and *catch*, as well as branching and looping can be used to control the flow of complex programs.

The syntax is:

throw exception

where the exception can be a string, number, object, etc.

Consider the following data entry application.



Upon entering an age and clicking on *Enter*, the application will confirm the adult status of the user, or otherwise. If an inadmissible age is entered, e.g., a negative value, a relevant exception shall be thrown. The exception shall be assigned to a variable which shall be written to the web page and displayed.

The code for the web page is as follows:

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

<script>

//this function fires when the Enter button is clicked on
function myFunction()
{
//declare the variable to hold the exception
```



```
//this variable will hold the exception and display it on the web page as part of a string
var y=document.getElementById("txt");

//initialize the variable
y.innerHTML="";

try
    {

        //the exception will be thrown based on the following age conditions
        //that are implemented using if statements
        var x=document.getElementById("age").value;

        if(x=="")    throw "You did not enter an age";
        if(isNaN(x)) throw "You did not enter a valid a number";
        if(x>=18)    throw "Adult status confirmed";
        if(x>0 && x<18) throw "Not an adult";
        if(x<0)      throw "You entered a negative value";
    }

//if an exception is thrown from the try block, do the following that is specified in the catch block
//note that the error object from the try will be assigned the relevant throw value
catch(err)
        {
            //assign the exception plus other string elements
            //to the variable for display on the web page
            y.innerHTML="Error: " + err + ".";
        }
}//this is the close of function

</script>

//web page set up
<p>Enter your age:</p>

//the text box
<input id="age" type="text">
```



```
//the click button
<button type="button" onclick="myFunction()">Enter</button>

//the text to display in the paragraph from the y variable
<p id="txt"></p>

</body>
</html>
```

Results are as follows:

Enter your age:

[ ] Enter

Error: You did not enter an age.

Enter your age:

[3r] Enter

Error: You did not enter a valid a number.

Enter your age:

[34] Enter

Error: Adult status confirmed.

Enter your age:

16 Enter

Error: Not an adult.

Enter your age:

-19 Enter

Error: You entered a negative value.

6.6 Debugging *JavaScript*

HTML and *JavaScript* scripts should be reviewed and thoroughly tested frequently as they are being developed. Web browsers do not throw error messages in a similar fashion and extent as a typical programming language, e.g., *Visual Basic*. It is the web developer's responsibility to frequently test the code and address problems as they arise. Verify or otherwise that the browser displays the web page content correctly and executes the scripts as intended. Test your codes and scripts frequently, block by block, line by line, tag by tag, rather than writing the entire code before testing it. In the latter scenario, it will be much more difficult to identify and isolate any problems.

After uploading your files to a web host it is important that the web pages and scripts be tested. For example, images should be checked for proper display. Forms and controls such as buttons, check boxes, and combo boxes should be clicked on to verify that the relevant scripts associated with them are executed and produce valid results. Links should be clicked on to verify that they open, and open to the correct location(s). Links and scripts that were set up to reference files in other folders will typically need to be reconfigured to reflect the path(s) to the file(s) and folder(s) as it now exists on the web hosting server.

6.7 Web Browser Compatibility

It is pertinent to note that different web browsers (and versions thereof), have some slight differences in the way they interpret and execute some commands. This must be kept in mind in the development and testing of your applications. In this course series, all codes were developed and tested with the *Internet Explorer* (version 10) web browser by *Microsoft*. It is the programmer's responsibility to test that the codes will work as intended on other web browsers.

6.8 Getting Help

There is currently an abundance of help information on *JavaScript*, web page design, and HTML, particularly on the web. These include official (peer-reviewed) and unofficial sources, websites, academic work, professional presentations, tutorial videos (YouTube, etc.), user groups, online forums, downloadable code snippets, etc., etc. Typing a *JavaScript* topic in a search engine will typically yield hundreds if not thousands of results. It is strongly recommended that all codes developed be tested thoroughly before deployment.

7. CONCLUSION

This course has presented a broad overview of fundamental concepts and principles of the *JavaScript* scripting language used for the development of web-based applications. The topics were presented with practical examples from situations encountered by practicing engineers and scientists.

In this course the topics, conditional statements, message boxes and alerts, looping structures, *JavaScript* objects, and error handling were covered in detail. Examples from engineering and other fields were used to illustrate and demonstrate the concepts and methods learned in this class. Two mini-projects were used to demonstrate these programming concepts and methods in a real-life web application.

This course has enabled participants to identify situations where web-based programming is relevant and will be of advantage to the practicing professional. Practitioners are strongly encouraged to look out for situations in their domains of expertise where web-based programming solutions are applicable and will be of benefit to their work and their organization.

Web development as well as web-based programming require a careful and meticulous approach and can only be mastered and retained by practice and repetition.

Good Luck and Happy Programming.

REFERENCES

ECMA International. (2011, June). *Standard ECMA-262 ECMAScript Language Specification Edition 5.1*. Retrieved March 26, 2014, from ECMA International: http://www.ecma-international.org/publications/standards/Ecma-262.htm

FunctionX Inc. (2011a). *Hypertext Markup Language*. Retrieved December 21, 2013, from FunctionX Tutorials: http://www.functionx.com/html/index.htm

FunctionX Inc. (2011b). *JavaScript Tutorial*. Retrieved March 21, 2014, from FunctionX Tutorials: http://www.functionx.com/javascript/index.htm

Microsoft. (2014a). *Error Object (JavaScript)*. Retrieved May 5, 2014, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/ie/dww52sbt(v=vs.94).aspx

Microsoft. (2014b). *JavaScript Functions*. Retrieved April 1, 2014, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/6fw3zxcx(v=vs.94).aspx

Microsoft. (2014c). *Special Charachters (JavaScript)*. Retrieved April 5, 2014, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/2yfce773(v=vs.94).aspx

Mozilla. (2014). *Array-JavaScript*. Retrieved May 1, 2014, from Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

quirksmode.org. (2014). *JavaScript - Cookies*. Retrieved May 1, 2014, from quirksmode.org: http://www.quirksmode.org/js/cookies.html

w3schools.com. (2014a). *JavaScript and HTML DOM Reference*. Retrieved 5 5, 2014, from w3schools.com: http://www.w3schools.com/jsref/default.asp

w3schools.com. (2014b). *JavaScript Cookies*. Retrieved May 1, 2014, from w3schools.com: http://www.w3schools.com/js/js_cookies.asp

World Wide Web Consortium. (2014). *HTML5*. Retrieved Aprl 9, 2014, from World Wide Web Consotium (W3C): http://www.w3.org/TR/2014/CR-html5-20140204/

Images were all drawn/-prepared by K. Ofosu

