

Introduction to JavaScript

Presented by: **Tim Kaufman**

INTRODUCTION

This summit session is for 4D Developers who have no prior experience with the JavaScript programming language. This introductory course will teach you the fundamentals of JavaScript programming.

1.0 – WHAT IS JAVASCRIPT

JavaScript is a programming language mainly used for web development. There is a good chance that, as you browse the web, you will run into more than a few pages where JavaScript is being executed. The code itself is either embedded in the HTML of the page or referenced in the page and stored externally. What can you do with JavaScript that you can't do with HTML? You can create dynamic Web documents, rewriting a page's content on the fly, responding to certain events or data. You can assert more control over user interaction, for example guiding a user in filling out a very complex form, prompting when input is done incorrectly, before the form is even submitted to the server.

1.1 – History

JavaScript was introduced in the mid 1990s by Netscape. After going through a few name changes, the name JavaScript stuck for the simple reason that an unrelated programming language, Java, was gaining momentum and popularity at the time. Marketing decided to ride this momentum and the name JavaScript was finalized. This naming has caused confusion, but remember: JavaScript is not an expansion of Java, or vice versa. They are two different languages that happened to be similarly named.

1.2 – Standardization

Other browsers besides Netscape soon began to support flavors of JavaScript, so a need for standardization was in place. Netscape went to ECMA International for this and a document was drafted that described exactly how JavaScript should work. ECMAScript, as this standardization is called, describes the language itself and not its integration into a web browser. So you can consider JavaScript to be ECMAScript and some additional tools for dealing with web pages and web browsers.

1.3 – Where Can JavaScript Be Used?

JavaScript, or variants of ECMAScript (the standardized language derived from JavaScript), are commonly used in following environments:

- Client-side / Browser, including Ajax and Flash (Flash uses ActionScript, a dialect of ECMAScript)
- Server Side (for example, ASP) (ASP primarily uses VBscript but can also use JavaScript or Jscript)
- On the desktop (for example, Windows Scripting Host, Adobe Acrobat, Adobe Photoshop, Adobe AIR)

1.4 – What Can JavaScript Do?

JavaScript can be used to accomplish many tasks including:

- Dynamically change HTML on an already loaded page
- Respond to events (like a mouse rollover or a key press)
- Validate data in an HTML form (and ask the user to correct it) before submitting to the server
- Detect and correct for browser incompatibilities
- Create and read cookies
- Parse and Send XML requests (now an Ajax staple)
- Some web-based video games have been written in JavaScript

1.5 – Suggested Tools for Development

Almost every operating system today should come with the tools required to program in JavaScript. That is because JavaScript is a client-side scripting language; that being said the simplest of pre-requisites would be:

- A modern Web Browser
- A simple text editor

Even though the requirements to develop in JavaScript are relatively simple, installing a few specific applications will make development much easier. An updated installation of the Mozilla Firefox browser and Firebug extension can drastically cut down on debugging time. In addition, either Notepad ++ (for Windows users) or Text Wrangler (for Macintosh users) will help with the code highlighting when editing HTML and JavaScript files.

- Mozilla Firefox web browser
- Firebug extension for Firefox
- Text Wrangler or BBedit (for Macintosh users)
- Notepad ++ (for Windows users)

Firefox can be found at:

<http://www.getfirefox.com/>

Firebug can be found at:

<http://www.getfirebug.com/>

Text Wrangler (mac) can be found at:

<http://www.barebones.com/products/textwrangler/>

Notepad++ (windows) can be found at:

<http://notepad-plus.sourceforge.net/>

Note: Although the combination of Firefox and Firebug make for a powerful debugging suite, it is best practice to constantly test your code in all of browsers (and platforms) during development.

2.0 – BASIC JAVASCRIPT

Let's look at the some of the Basics of JavaScript, including the Syntax, Values, Variables, Operators, and Control Flow.

2.1 – Syntax

Syntax is probably the most important aspect of any programming language. The syntax of JavaScript is a set of rules that defines what constitutes a valid program in the JavaScript language.

When writing JavaScript you can store your code either inline with the HTML code or you can store your JS code in a separate .JS file and then reference the JS file from your code.

Example of inline JavaScript:

```
<HTML>
<HEAD>
<script>
<!--
// this is a comment, comments are preceded by two forward-slashes
var test = 25;
function myAlert(x) {
    // this comment is inside of my function
    alert(x); // displays an alert
}
myAlert(prompt('What is a number','')+test); // call the myAlert function
-->
```

```
</script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

When writing JavaScript inline, you place your code within a `<script>` and `</script>` tag. Inside of the `<script>` tags you also want to encapsulate your JS code with HTML comments (`'<!--'` and `'-->'`), to avoid your code being processed by the web server.

Inline JavaScript is probably the most frequently used, especially among novice JavaScript programmers and web-scripters.

Here is an example of using an external JavaScript file:

```
Example of using an external JavaScript
<HTML>
<HEAD>
<script src="js/myLibrary.js"></script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

As you can see in the example above, the HTML file looks much cleaner when the JavaScript is stored in an external file. The `SRC=` attribute of the `<script>` tag has the relative path to my .js file.

2.2 – Values

To create a value in JavaScript, one must merely invoke its name and assign it a value. You do not have to declare its type as there is no need for variable typing in JavaScript; this is very convenient and time saving.

Example declaration:

```
var x = "Feels like summer started early this year";
```

The basic types of values are: Numbers, Strings, Booleans, Null and Undefined values.

Numbers in JavaScript are "double-precision 64-bit format IEEE 754 values", according to the spec. This has some interesting consequences. There's no such thing as an integer in JavaScript, so you have to be a little careful with your arithmetic if you're used to math in C or Java. Watch out for stuff like:

```
0.1 + 0.2 = 0.30000000000000004
```

Numbers

Values of the type number are represented by numeric values like so:

```
var x = 144;
```

Numeric values can also be represented in binary format like so:

```
var x = 01010101;
```

NOTE: When writing a numeric value in binary format the number must start with a 0 and contain only 1's and 0's.

Fractional numbers are written using a dot like so:

```
var x = 285.5;
```

Octal numbers are supported:

```
var x = 0377;
```

Hexadecimal numbers are supported:

```
var x = 0xFF;
```

Note: When writing hexadecimal numbers, the letters A-F may be upper- or lowercase.

Scientific Notation is also supported:

```
var x = 5.965e8;
```

Strings

Strings can be a combination of numbers and letters. Strings can be thought of similarly as Text variables in 4D.

```
var x = "this is a string";
```

Boolean

Boolean values are simply true or false

```
var x = false;
```

Note: In JavaScript true and false values must be written lower case

Null

Null is an assignment value that can be assigned to a variable to represent that the variable has no value. Null differs from Undefined in that the value is defined, although it has been defined as having no value. Consider the following;

```
Examples/Example_2-1_null.html
<HTML>
<HEAD>
<script>
<!--
var test = null;
alert('test = ' + test); // displays null
alert('typeof test = ' + typeof test); // displays object
-->
</script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

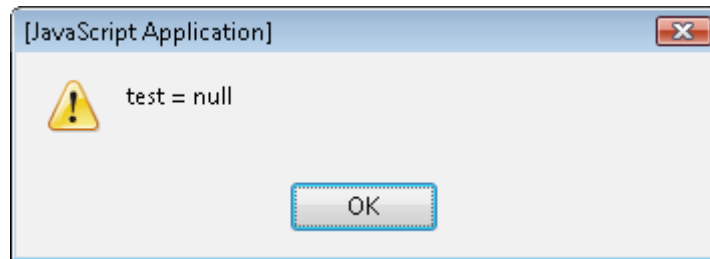
The above code snippet first declares a variable named test, and then assigns the value of null to it:

```
var test = null;
```

The next line of code displays an alert, telling us what the variable test resolves to:

```
alert('test = ' + test); // displays null
```

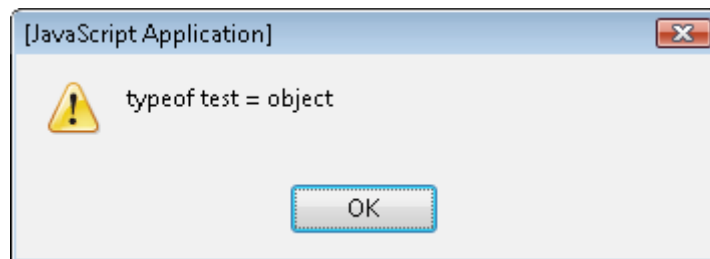
When that code is run the browser alerts us with the following message:



The next line of code displays an alert, telling us what type of variable test is:

```
alert('typeof test = ' + typeof test); // displays object
```

When the code is ran the browser alerts us with the following message:



Undefined

Undefined means a variable has been declared but has not yet been assigned a value. Consider the following;

```
Examples/Example_2-1_undefined.html
<HTML>
<HEAD>
<script>
<!--
var test;
alert('test = ' + test); // displays undefined
alert('typeof test = ' + typeof test); // displays undefined
-->
</script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

The above code snippet first declares a variable named test, but does not assign a value to it:

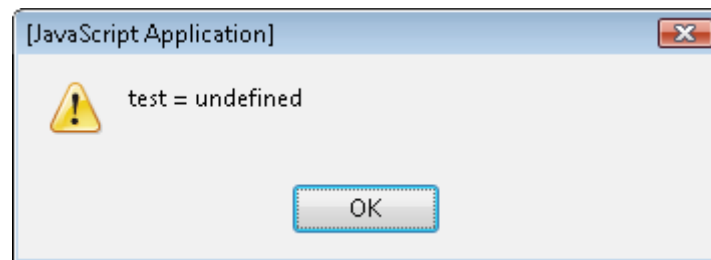
```
var test;
```

Note: The only difference between this example for Undefined and the earlier example for Null is that with Null a value was assigned to the variable.

The next line of code displays an alert, telling us what the variable test resolves to:

```
alert('test = ' + test); // displays undefined
```

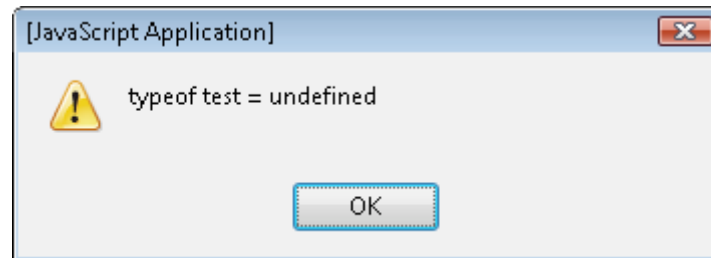
When that code runs, the browser alerts us with the following message:



The next line of code displays an alert, telling us what type of variable test is:

```
alert('typeof test = ' + typeof test); // displays undefined
```

When the code runs, the browser alerts us with the following message:



2.3 – Variables

Variables in JavaScript have no user assigned types associated with them, and any value may be stored in any variable. Variables can be declared using the var keyword. These variables are lexically scoped and once they are declared, they may be accessed anywhere inside the function they were declared in. Variables declared outside of any function, and variables first used within functions without being declared with the 'var' keyword, are global once the function runs.

Note: Variable names are case sensitive. The variable 'a' is not the same as the variable 'A'.

```
Examples/Example_2-3_Variables.html
a = 0; // a global variable
var b = "Howdy!"; // another global variable

function c(){
  g = "geek"; // a global variable
  var f = "inside of c function"; // local variable
```

```

        return g + f;
    }

    h = c();

    document.write("a = " + a + "<br>");
    document.write("g = " + g + "<br>");
    document.write("h = " + h + "<br>");

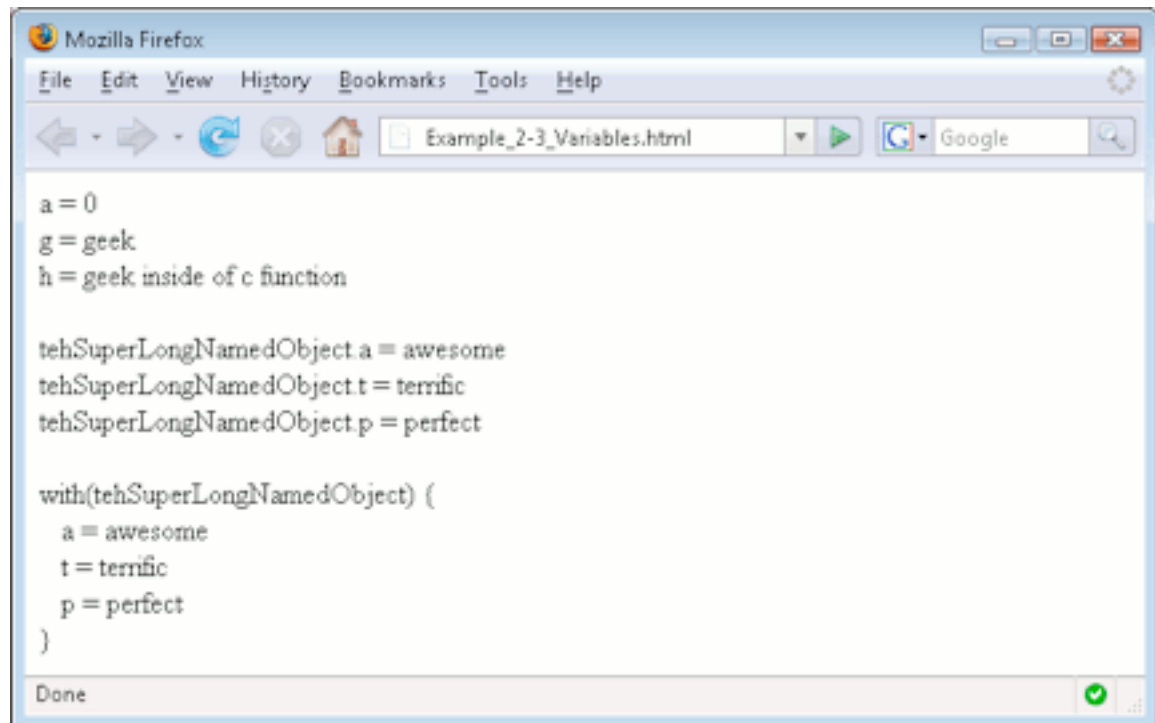
    var tehSuperLongNamedObject = {
        p: "perfect",
        t: "terrific",
        a: "awesome"
    }

    document.write("<br>");
    document.write("tehSuperLongNamedObject.a="+tehSuperLongNamedObject.a+"<br>");
    document.write("tehSuperLongNamedObject.t="+tehSuperLongNamedObject.t+"<br>");
    document.write("tehSuperLongNamedObject.p="+tehSuperLongNamedObject.p+"<br>");

    with(tehSuperLongNamedObject){
        document.write("<br>");
        document.write("with(tehSuperLongNamedObject) {<br>");
        document.write("&nbsp;&nbsp;&nbsp;a = " + a + "<br>");
        document.write("&nbsp;&nbsp;&nbsp;t = " + t + "<br>");
        document.write("&nbsp;&nbsp;&nbsp;p = " + p + "<br>");
        document.write("}");
    }
}

```

The above code snippet above produces the following results.



In the example above, the variable 'f' is local to the 'c' function because it was declared inside of the 'c' function using the var keyword. On the other hand, even though the variable 'g' was first used within the 'c' function, it is a global variable because it was not declared with the var keyword. The variable 'g' will therefore be available to the rest of the script once the function has run. The example above also shows two ways of dealing with objects; one being writing out the full name (e.g. tehSuperLongNamedObject.t) and the other being to use a with statement. This will be discussed more in the Scope Chain subsection.

Variable names can be almost every word, but they cannot include spaces. Digits can be part of the variable name but the name must not start with one (e.g., catch22 is a valid variable name but 4D is not). The '\$' and '_' characters can be used in variable names as if they were letters, so \$_\$ is a valid variable name as is _4D and \$4D.

\$4D	"Valid name"
_4D	"Valid name"
a	0
b	"Howdy!"
g	"geek"
h	"geek inside of c function"
tehSuperLongNamedObject	Object p=perfectt=terrific a=awesome
a	"awesome"
p	"perfect"
t	"terrific"
c	c()
prototype	Object

In addition to these rules, there are also some special keywords that cannot be used as variable names:

```
abstract boolean break byte case catch char class const continue debugger
default delete do double else enum export extends false final finally float
for function goto if implements import in instanceof int interface long
native new null package private protected public return short static super
switch synchronized this throw throws transient true try typeof var void
volatile while with
```

Variable Types

The 4D developer is probably used to having to type their variables. JavaScript supports dynamic typing so types are associated with values, not variables. For example, a variable x could be bound to a number, and then later rebound to a string.

Scope Chain

Scope can be thought of as a way to determine the context of where you are and what is around us... This is important because in JavaScript the scope plays a valuable role.

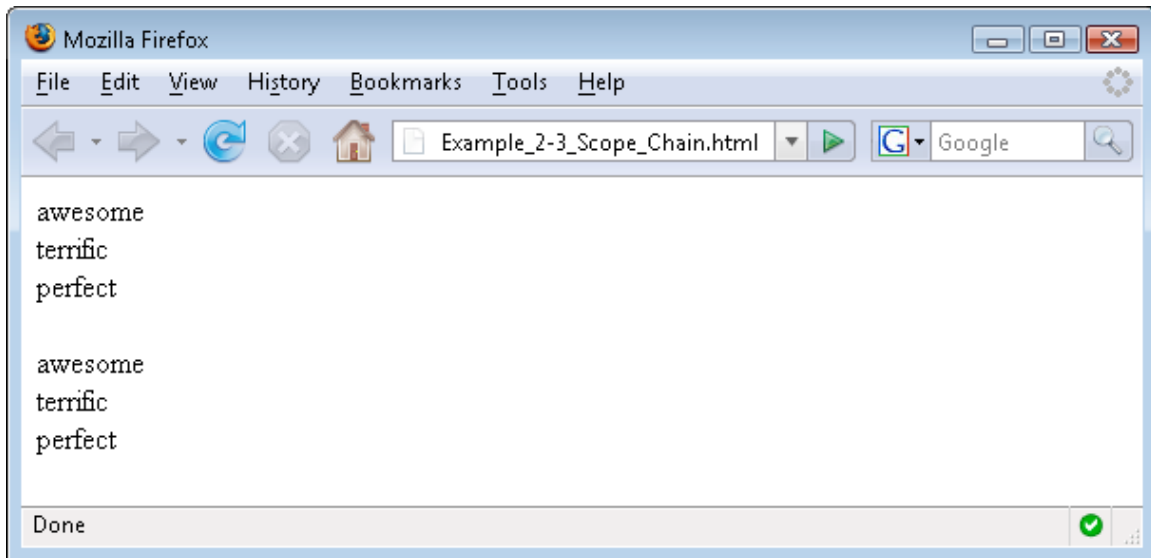
In JavaScript, if you declare a local variable within a function or an object, it will not be available outside of that function or object. In the previous section our example declared two variables named 'a', one was global to the whole script, the other was a property of an object... Just referencing the variable name 'a' will surely use the global variable, but what if we wanted to reference the property of the object? To do this would be objectName.propertyName – but what if we had multiple properties we needed to reference and didn't want to type out the full objectName multiple times? Well, JavaScript has a nice little feature, called with that can augment the scope chain, to put an object at the top of the scope to be searched first. Consider the following:

```
document.write(tehSuperLongNamedObject.a+"<br>");
document.write(tehSuperLongNamedObject.t+"<br>");
document.write(tehSuperLongNamedObject.p+"<br>");
```

Versus

```
with(tehSuperLongNamedObject){
    document.write(a + "<br>");
    document.write(t + "<br>");
    document.write(p + "<br>");
}
```

It is not required to use the with keyword, but it can be useful in certain situations. It is important to note, both produce the same results:



2.4 – Operators

This section covers JavaScript operators including math operators, bitwise operators, and logical operators. Some advanced math functions are also discussed in this section.

Mathematical Operators

Mathematical operations are probably the most commonly used operators in JavaScript. Here is a list of the commonly used mathematical operators.

Operator	Meaning	Explanation
+	Addition	Adds to numbers or appends two strings. If more than one type of variable is used (e.g., number + string) the result will be a string.
-	subtraction	Subtracts the second number from the first
*	Multiplication	Multiplies two numbers
/	Division	Divides the first number from the second
%	Modulus	Divide the first number by the second and return the remainder
++	Increment	Increment the number by 1
--	Decrement	decrement the number by 1

The '+' sign is overloaded; it is used for string concatenation as well as mathematical addition and is also used to convert strings to numbers. It also has a special meaning when used in regular expressions. Consider the following:

```
Examples/Example_2-4_Addition.html
<HTML>
<HEAD>
<script>
<!--
// Concatenate 2 strings
```

```

var x = 'This';
var y = ' and that';
alert(x + y); // displays 'This and that'

// Add two numbers
var a = 2;
var b = 6;
alert(a + b); // displays 8

// Adding a string and a number results in concatenation
alert( a + '2'); // displays 22

// Convert a string to a number
var c = '4'; // c is a string (the digit 4)
alert( c + a); // displays 42
alert( +c + a); // displays 6

-->
</script>
</HEAD>
<BODY>
</BODY>
</HTML>

```

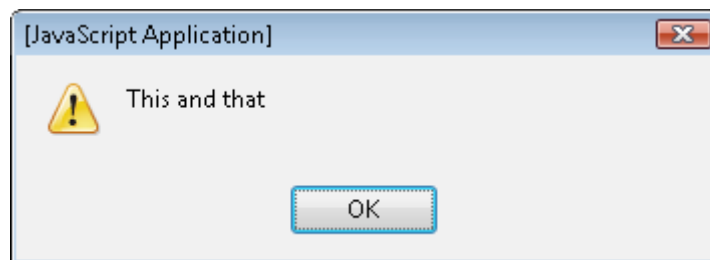
In the code snippet above, various uses of the '+' operator are shown. In the first part, it is shown that when x and y are strings, the '+' operator will concatenate the arguments.

```

// Concatenate 2 strings
var x = 'This';
var y = ' and that';
alert(x + y); // displays 'This and that'

```

The code snippet above results in the following alert box:



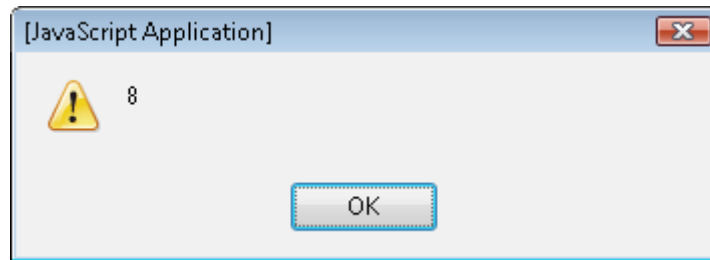
The next part of the code snippet shows that using the '+' operator on two numbers will indeed mathematically add the two numbers together

```

// Add two numbers
var a = 2;
var b = 6;
alert(a + b); // displays 8

```

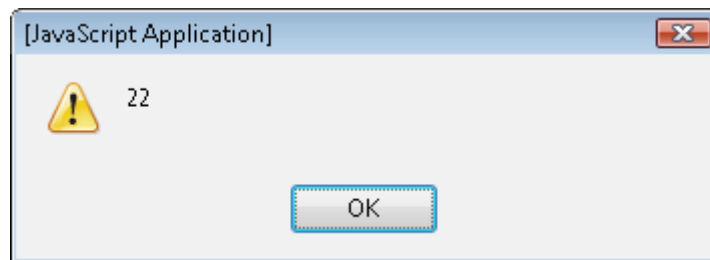
The code snippet above results in the following alert box:



The next part of the code snippet shows that using the '+' operator on one string and one number will result in the two being concatenated together.

```
// Adding a string and a number results in concatenation  
alert( a + '2'); // displays 22
```

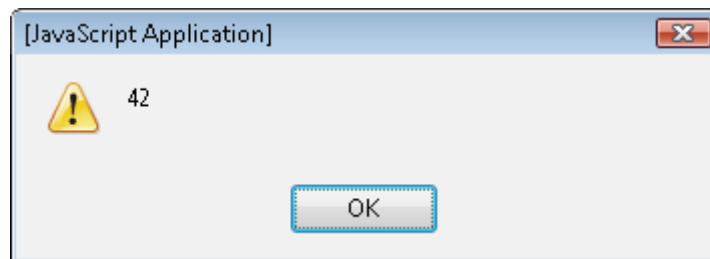
The code snippet above results in the following alert box:



In the last part of the code snippet the variable 'c' is defined as the digit '4'. Because the value is surrounded by quotes it is a string.

```
// Convert a string to a number  
var c = '4'; // c is a string (the digit 4)  
alert( c + a); // displays 42
```

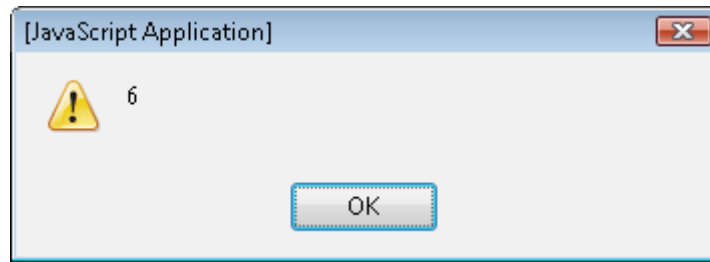
The code snippet above results in the following alert box:



The '+' operator can be used directly to the left of the variable name to try to convert it to a number. If the variable is unable to be converted to a number, the result will be a NaN. This would occur, for example, if the value was a string.

```
alert( +c + a); // displays 6
```

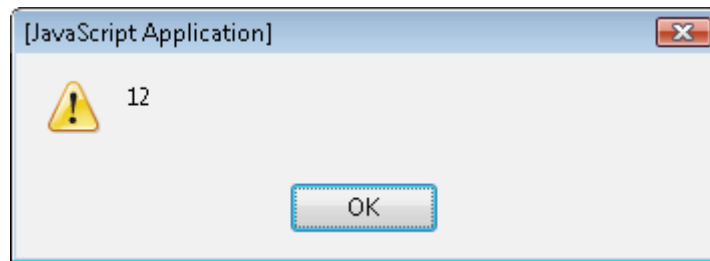
The code snippet above results in the following alert box:



Here is an example of using the multiplication operator:

```
Examples/Example_2-4_Multiplication.html
<HTML>
<HEAD>
<script>
<!--
var x = 3;
var y = 4;
alert(y*x);
-->
</script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

The above snippet of code declares two variables, 'x' as 3 and 'y' as 4. After declaring the variables, an alert is called displaying the product of 'x' multiplied by 'y'; as shown in the following screenshot:



In addition to the mathematical operators in the table above, there are also mathematical functions already defined in JavaScript. Here is a list of the more advanced mathematical functions in JavaScript.

Method	Syntax	Explanation
abs	Math.abs(x)	Returns the absolute value of x
acos	Math.acos(x)	Returns the arccosine (in radians) of x
asin	Math.asin(x)	Returns the arcsine (in radians) of x
atan	Math.atan(x)	Returns the arctangent (in radians) of x
atan2	Math.atan2(y, x)	Returns the arctangent of the quotient y divided by x
ceil	Math.ceil(x)	Returns the smallest integer greater than or equal to x
cos	Math.cos(x)	Returns the cosine of x
exp	Math.exp(x)	Returns E^x where x is the argument and E is Euler's constant, the base of the

		natural logarithms
floor	Math.floor(x)	Returns the largest integer less than or equal to x
log	Math.log(x)	Returns the natural logarithm (base E) of x
max	Math.max(a,b,c,...x)	Returns the largest of zero or more numbers
min	Math.min(a,b,c,...x)	Returns the smallest of zero or more numbers
pow	Math.pow(base,exp)	Returns base to the exp power. That is baseexp
random	var x = Math.random()	Returns a pseudo-random number in the range of [0,1) – that is, 0 (inclusive) and 1 (exclusive). The random number generator is seeded from the current time, as in java
round	Math.round(x)	Returns the value of a x rounded to the nearest integer
sin	Math.sin(x)	Returns the sine of x
sqrt	Math.sqrt(x)	Returns the square root of x – if the value of x is a negative number a NaN is returned
tan	Math.tan(x)	Returns the tangent of x

*Note: -Trigonometric functions assume that the argument is in radians, not degrees.
-NaN means "Not a Number."*

When using several Math constants and methods in a section of code it is often times more convenient to use the with statement like so:

```
Example
var a, x, y;
var r = 10;
with (Math) {
    a = PI * r*r;
    y = r*sin(theta);
    x = r*cos(theta);
}
```

As opposed to the following snippet written without the with statement:

```
var a, x, y;
var r = 10;
a = Math.PI * r*r;
y = r*Math.sin(theta);
x = r*Math.cos(theta);
```

Note: The with keyword is used to extend the scope chain for a statement as described in section "2.1 – Syntax"

Logical Operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified

operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value.

Operator	Meaning	Example	Explanation
&&	AND	expr1 && expr2	Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
	OR	expr1 expr2	Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, returns true if either operand is true; if both are false, returns false.
!	NOT	expr1 ! expr2	Returns false if its single operand can be converted to true; otherwise, returns true.

Comparison Operators

The operands can be numerical or string values. Strings are compared based on standard lexicographical ordering, using Unicode values. JavaScript has both strict and type-converting equality comparison. For strict equality the objects being compared must have the same type and:

Two strings are equal when they have the same sequence of characters, same length, and same characters in corresponding positions

Two numbers are strictly equal when they are numerically equal (have the same number value). NaN is not equal to anything, including NaN. Positive and negative zeros are equal to each other

Two Boolean operands are strictly equal if they are both true or false

Two objects are strictly equal if they refer to the same Object

Null and Undefined types == (but not strictly ===)

The following table describes the comparison operators:

Operator	Meaning	Explanation
==	Is equal to	If the two operands are not of the same type, JavaScript converts the operands then applies strict comparison. If either operand is a number or a Boolean, the operands are converted to numbers; if either operand is a string, the other one is converted to a string
===	Strict Is equal to	Returns true if the operands are strictly equal with no type conversion
!=	Not equal to	Returns true if the operands are not equal. If the two operands are not the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison
!==	Strict not equal to	Returns true if the operands are not equal and/or not

		of the same type
>	is greater than	Returns true if the left operand is greater than the right operand
<	is less than	Returns true if the left operand is less than the right operand
>=	is greater than or equal to	Returns true if the left operand is greater than or equal to the right operand
<=	is less than or equal to	Returns true if the left operand is less than or equal to the right operand

Special Operators

Here are some additional operators with specialized uses.

Operator	Meaning	Explanation
?	Conditional	The conditional operator is the only JavaScript operator that takes three operands. This operator is frequently used as a shortcut for the if statement. condition ? ifTrue : ifFalse;
,	Comma	The comma operator evaluates both of its operands and returns the value of the second operand. You can use the comma operator when you want to include multiple expressions in a location that requires a single expression. The most common usage of this operator is to supply multiple parameters in a for loop.

```
Examples/Example_2-4_Conditional.html
confirm("Question") ? alert("you clicked OK") : alert("you clicked
Cancel");
```

The code snippet above shows a conditional statement. The conditional statement in this example displays a confirmation box to the user, the confirmation box simply says “Question” and returns either true or false depending on whether the user clicks OK (true) or Cancel (false). If the user clicks OK, the confirm dialog will return true and the first condition will be ran alerting the user that “you clicked OK”. If the user clicks Cancel the confirm dialog will return false and the second condition will be ran alerting the user that “you clicked Cancel”.

Conditional statements do not need to be confirmation dialogs; you could write a conditional statement that checks a value of one variable and sets the value of another depending on the value of the first. Consider the following:

```
Examples/Example_2-4_Conditional2.html
var myFavoriteColorIsRed = (favoriteColor == 'red') ? true : false;
```

The code snippet above sets the variable myFavoriteColorIsRed to true if the variable favoriteColor is equal to ‘red’, otherwise the myFavoriteColorIsRed is set to false. Another way to write this would be:

```
Examples/Example_2-4_Conditional3.html
if(favoriteColor == 'red'){
    myFavoriteColorIsRed = true
}else{
    myFavoriteColorIsRed = false
}
```

As you can see, the conditional statement is considerably shorter and would take less time to write. Either way you write it, the result will be the same.

favoriteColor	"red"	favoriteColor	"blue"
myFavoriteColorIsRed	true	myFavoriteColorIsRed	false

2.5 – Control flow

Your methods will always use one of the three types of programming structures, regardless of the method's complexity. These three types control the execution's flow including the order of statements executed. They will be described individually in more detail below.

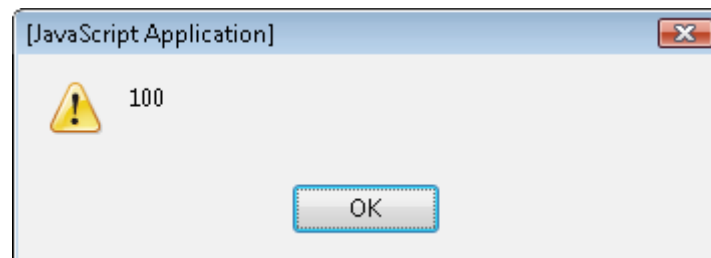
Sequential
Looping
Branching

Sequential

A sequential structure is in place when your code moves from one statement to another following a linear path from the top-most statement to the bottom. Here is an example of that.

```
Examples/Example_2-5_sequential.html
var x = 5;
var y = 20;
var z = x*y;
alert(z);
```

The above example sets the 'x' variable to 5 and the 'y' variable to 20. The 'z' variable is set to the product of 'x' and 'y' then an alert is displayed showing the value of 'z'.



Looping

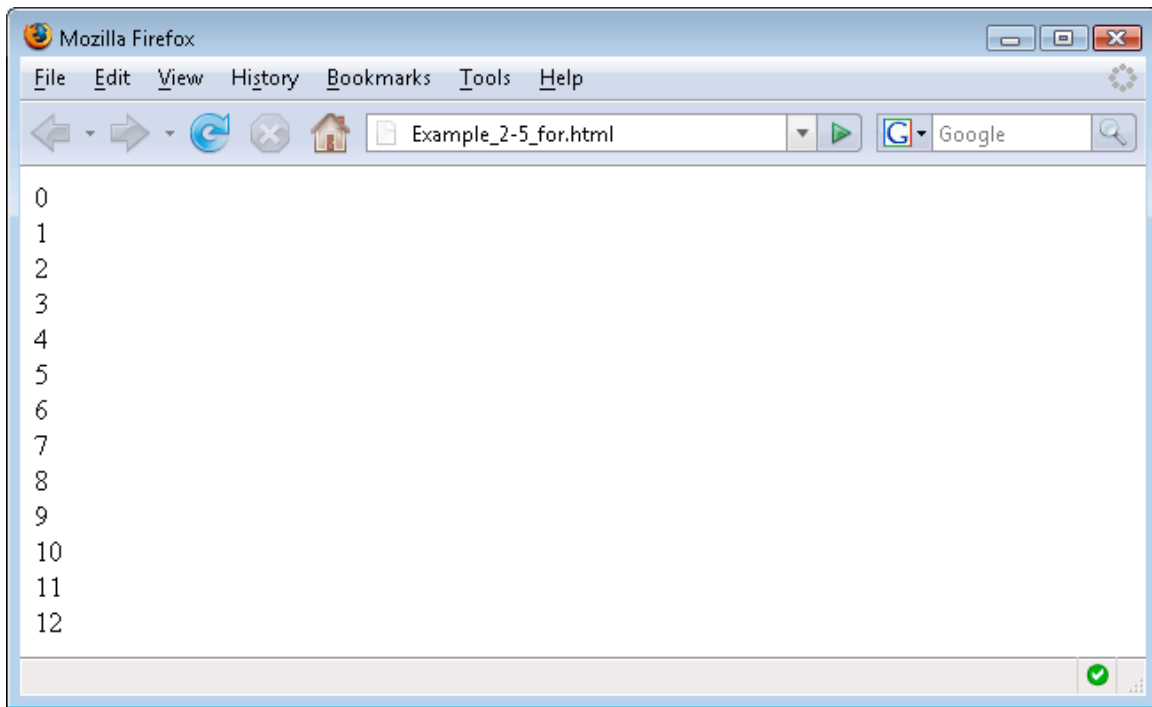
If you need to recursively repeat a sequence of statements more than once, Javascript provides three looping structures for you to use.

For

For loops are very useful when you need a block of code to repeat a certain number of times. Here are some examples of

```
Examples/Example_2-5_for.html
for(var myCounter = 0; myCounter < 13; myCounter++){
    document.write(myCounter+'<br>');
}
```


The above code will loop through the number 0 to 12 and write each one to the webpage on its own line.



```
Examples/Example_2-5_for_comma.html  
for(var counter1=0,counter2=9;counter1<10; counter1++,counter2--)  
document.write(counter1+"x"+counter2+"="+counter1*counter2+"<br>");
```

The code snippet above is a for loop. The loop statement is broken down into three parts separated by semi-colons. When broken up, the three parts are:

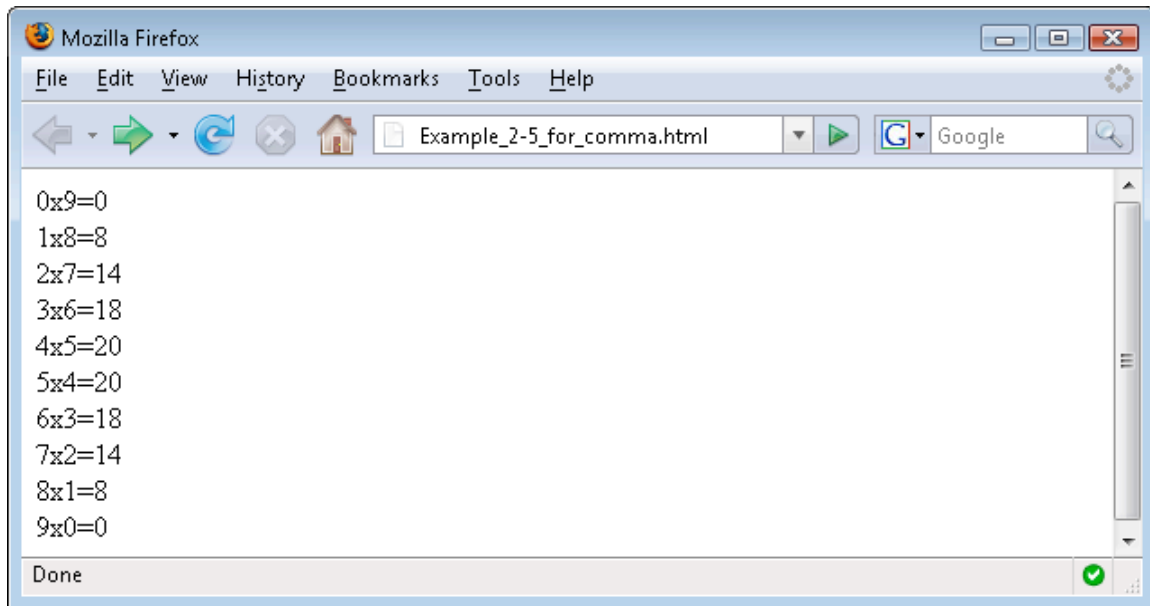
- var counter1=0, counter2=9;
- counter1 < 10;
- counter1++, counter2--

In the first part of this loop we set the variable 'counter1' to 0 and the variable 'counter2' to 9. In the second part of the loop statement, 'counter1 < 10', we loop for as long as the variable 'counter1' is less than 10. The last part of the loop statement, 'counter1++, counter2--' says for each iteration of the loop the variable 'counter1' will be incremented and the variable 'counter2' will be decremented.

The line immediately following the for statement will be repeated for the iterations specified in the for statement. If more than one line of code needed to be repeated it should be encapsulated with squiggly brackets. In this example the following line of code will be repeated:

```
document.write(counter1+"x"+counter2+"="+counter1*counter2+"<br>");
```

Which would result in the following page:

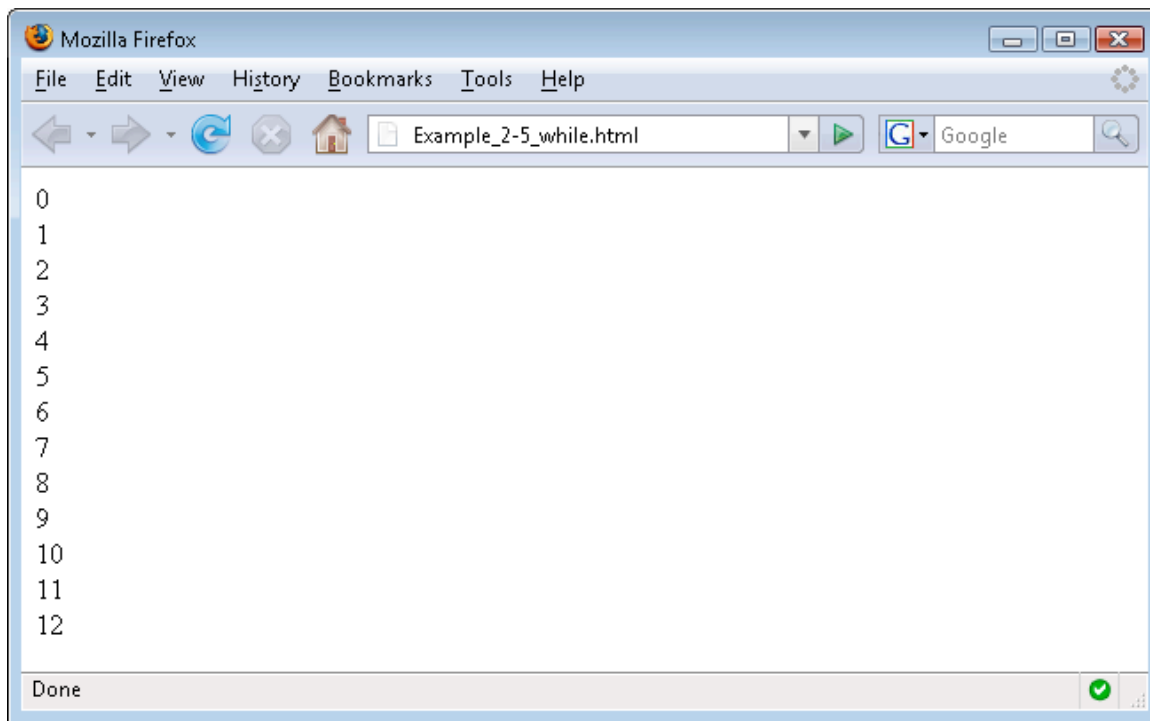


While

While loops are useful when you need a block of code to be repeated while a specified condition is true.

```
Examples/Example_2-5_while.html
var myCounter = 0;
while(myCounter < 13){
    document.write(myCounter+'<br>');
    myCounter++
}
```

The above code will loop through the number 0 to 12 and write each one to the webpage on its own line.

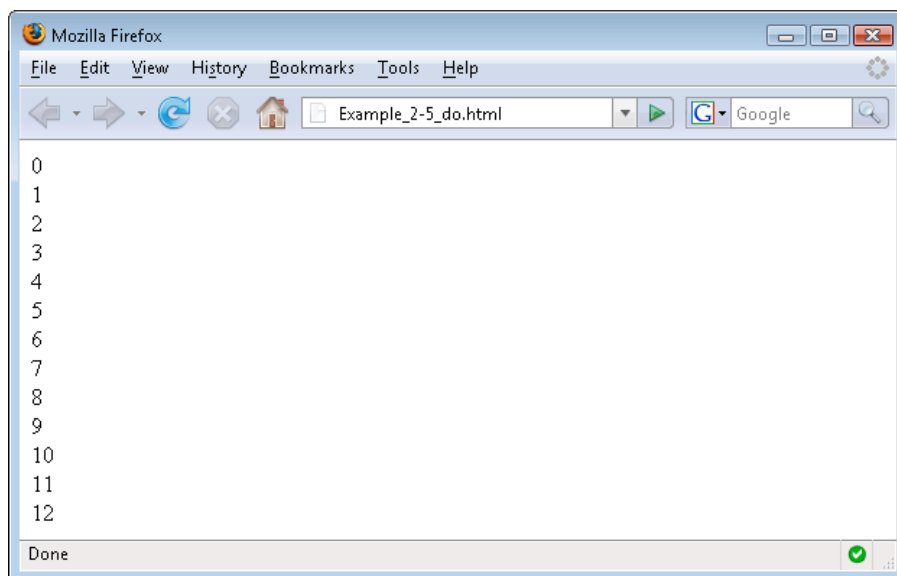


Do

A do loop is a variant of the while loop.

```
Examples/Example_2-5_do.html
var myCounter = 0;
do{
    document.write(myCounter+'<br>');
    myCounter++;
}
while (myCounter<13)
```

The above code will loop through the number 0 to 12 and write each value to the webpage on its own line.



Branching

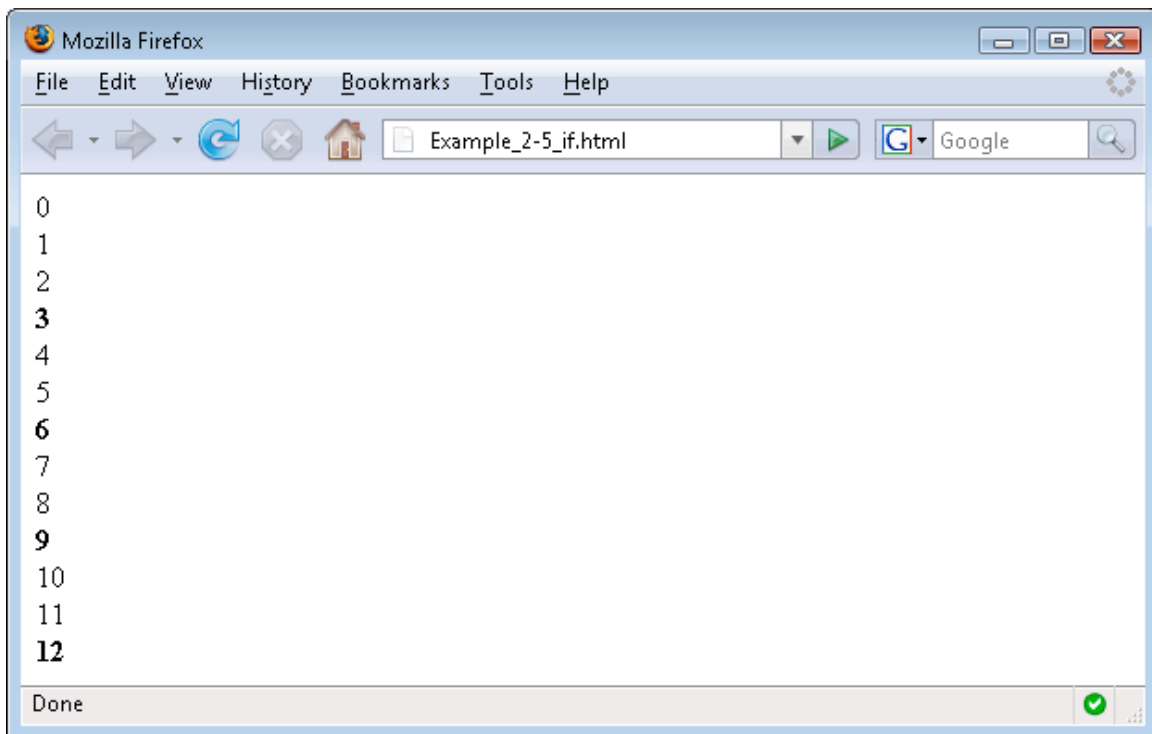
The last of the three structures is called a branching structure. Branching lets you test a condition and take different paths depending on the results of the condition. An example of a branching structure is an If...Else...End statement, which directs program flow accordingly.

If / Else / Else If / Break

If statements are generally used to check specific conditions and run code appropriate to the conditions present. Here are some examples of if statements written in JavaScript:

```
Examples/Example_2-5_if.html
for(var myCounter = 0; myCounter < 13; myCounter++){
    if(myCounter % 3 == 0 && myCounter != 0){
        document.write('<b>'+myCounter+'</b><br>');
    }else{
        document.write(myCounter+'<br>');
    }
}
```

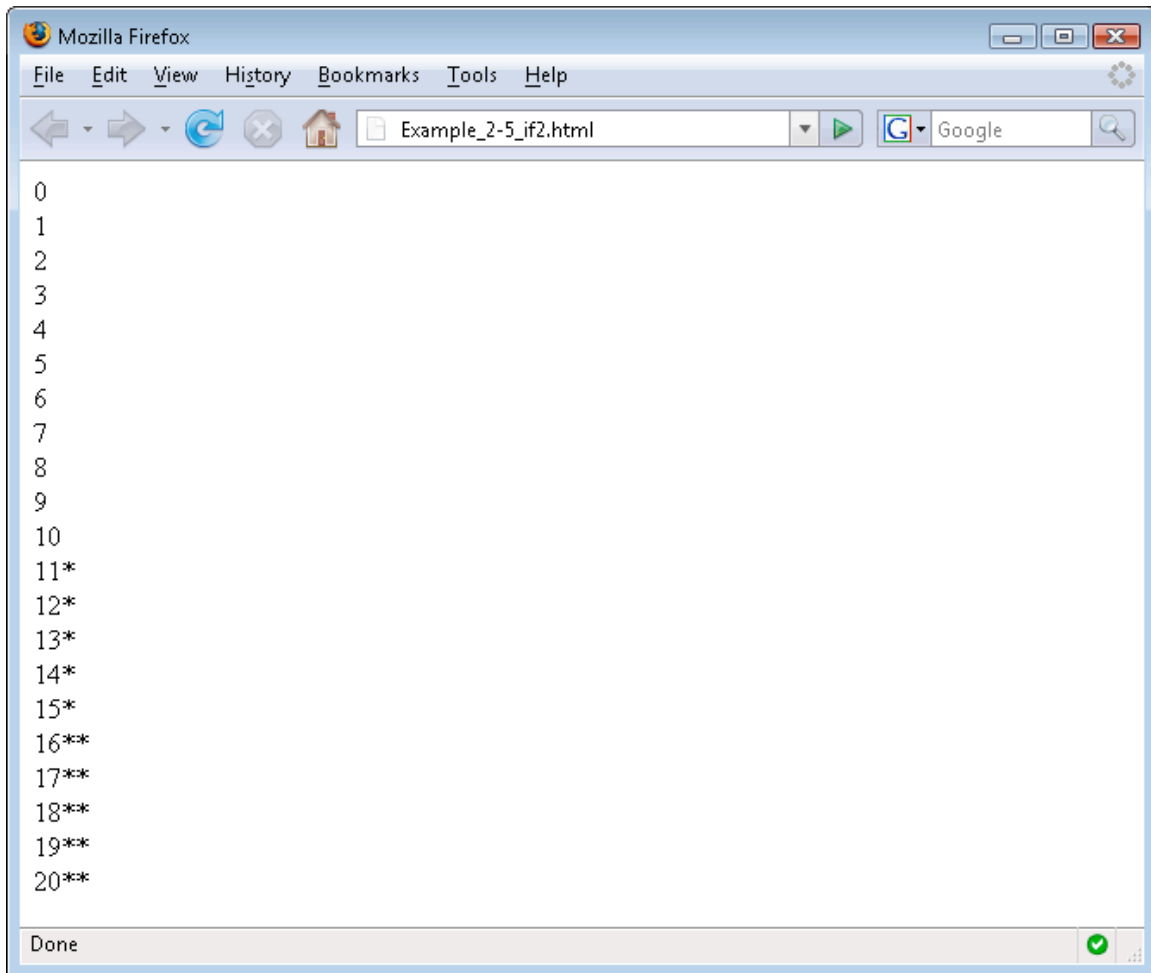
The code above will loop through the numbers 0 to 12 and write each of one to the webpage on its own line. The numbers that are cleanly divisible by 3 will be marked in bold (excluding 0).



The following example will loop through the numbers 0 through 20 and write each number to the webpage on its own line. The numbers 16 through 20 will have two asterisk next to it, the numbers 11 through 15 will have 1 asteroid next to them, and numbers 0 through 10 will be written normal.

```
Examples/Example_2-5_if2.html
for (var myCounter = 0; myCounter <= 20; myCounter++) {
    if (myCounter > 15)
        document.write(myCounter + "***" + "<br>");
    else if (myCounter > 10)
        document.write(myCounter + "*" + "<br>");
    else
        document.write(myCounter + "<br>");
}
```

The following image displays the output that would be visible on the webpage:



User Interaction

An important part of any program is user interaction. User interaction allows an application to behave in response to specific interactions with the user. As the developer, you are able to define these interactions. The basic types of user interaction are Alert, Prompt, and Confirm:

Alert

An alert is one way to display information to the user. This type of user interaction is normally used for errors and other informative messages where there is no decision to be made by the user.

```
Examples/Example_2-5_alert.html
```

```
alert("Hello World!");
```

The above line of code will display an alert message on the screen with the text "Hello World!" in the message box.



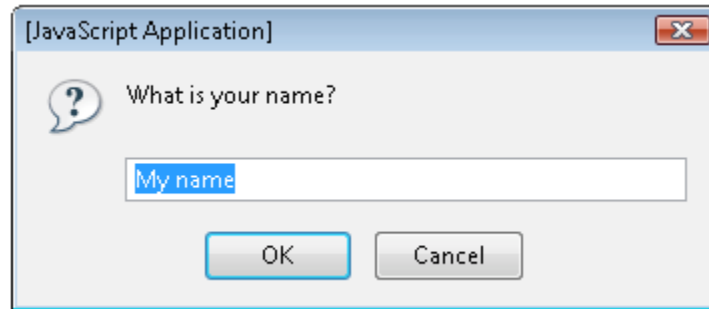
As you can see from the image above, an alert produces a message box with a single button (OK) to push.

Prompt

A prompt is used when you want a written response from the user, such as their name.

```
Examples/Example_2-5_prompt.html  
var myname = prompt("What is your name?", "my name");
```

The above code will prompt the user with a message “What is your name?” and will have a default value of “my name”. The information submitted will be available in the variable named myname. The dialog box will look similar to the following image in FireFox:



If the user clicks OK the myname variable will be set to whatever is entered into the prompt, if the user clicks CANCEL the myname variable will be null. If the user enters 3, the myname variable will be a string; to force it to be a number instead of a string you can place the + operator to the left of the prompt command like so:

```
var myname = +prompt("What is your name?", "my name");
```

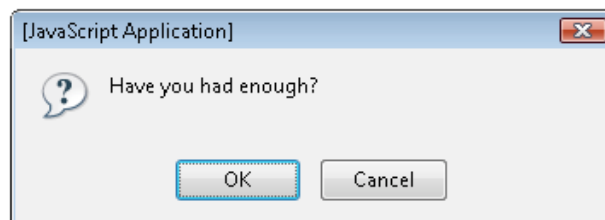
However, if you use this method and the user inputs something other than numbers the myname variable will be set to NaN. Also, with the prompt asking for your name and supplying a default value of “my name”, it would be a little strange to have your JavaScript coded to expect a number instead of a string. Either way, that is the procedure of how you do it.

Confirm

The confirm command is used to when you need the user to confirm an action or information.

```
Examples/Example_2-5_confirm.html  
var myanswer = confirm("Have you had enough?");
```

The above line of code produces a dialog box that asks “Have you had enough?” and has two buttons, an OK and a CANCEL button. The user’s action will be stored in the variable named myanswer. The dialog will look similar to the following image:



If the user clicks OK the myanswer variable will be set to true, if the user clicks CANCEL the myanswer variable will be set to false.

3.0 – FUNCTIONS

Being a 4D Developer, you are probably already familiar with writing methods. A function is very similar to a method in a sense that you write reusable code inside of functions, than call these functions elsewhere in your code. Functions can interact with each other, and even interact with the user. In this section we will explore writing functions and passing arguments to our functions.

3.1 – Writing a function

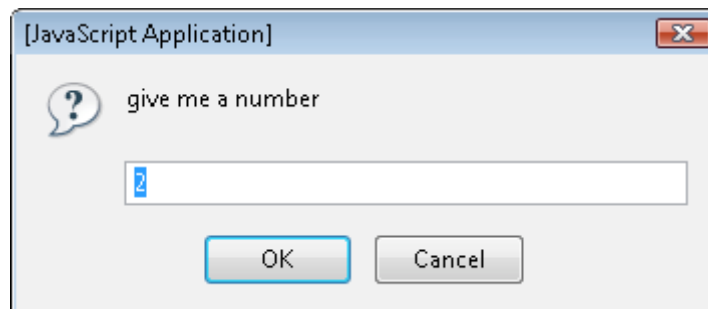
Writing a function in JavaScript is pretty simple. You start with the keyword function and follow it with the name of your function. Following the function name is the parenthesized arguments your function is expecting followed by an open squiggly bracket. All of your code goes after this open squiggly bracket, and is followed by a close squiggly bracket.

```
Examples/Example_3-1_Writing_a_Function.html
function mytestfunction(){
    var arg1 = prompt("give me a number","2");
    var arg2 = prompt("give me another number","3")
    sum = arg1 + arg2;
    alert(sum)
}
```

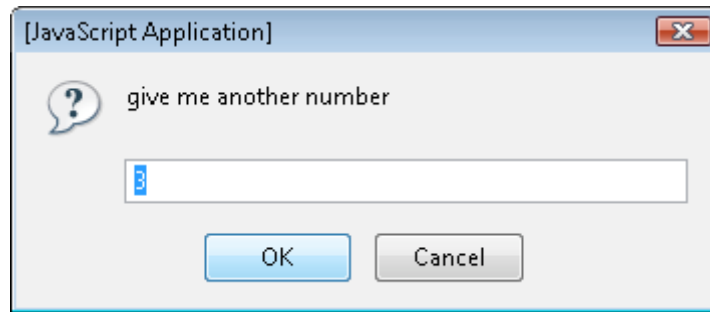
Note: Pages that are abundant with alerts and confirms suffer from poor design. Alert/Confirm/Prompt boxes block the entire browser and "surfing" experience. They should only be used if it's critical information, such as "are you sure you want to ERASE ALL DATA in this form?"

Although parenthesized arguments were mentioned in the passage preceding this example, our function in the example above does not expect any arguments (section 3.2 will go over passing arguments to a function). Instead, this function prompts the user for input during execution. Let's look more closely at what the code snippet above does.

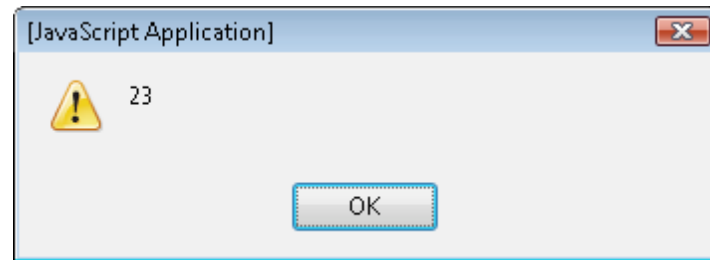
First it prompts the user with the message "give me a number" and presents the user with '2' as the default value. When the user clicks OK the value entered will be set to the arg1 variable. Here is a screenshot of the dialog box:



The function then prompts the user with the message "give me another number" and presents the user with '3' as the default value. When the user clicks OK the value entered will be set to the arg2 variable. Here is a screenshot of the dialog box:



The function then tries to add the two variables together using the '+' operator and then display the result using an alert box. Here is the result when the default values are used:



You are probably aware that the value displayed in the alert box is not what we were expecting. This is because the values were interpreted as strings so the '+' operator concatenated the two variables together.

Let's take what we learned about the '+' operator in the previous sections and modify the test function above to treat our variables as numbers instead of strings.

Answer:

```
function mytestfunction(){
    var arg1 = prompt("give me a number",2);
    var arg2 = prompt("give me another number",3)
    sum = +arg1 + +arg2;
    alert(sum)
}
```

Or

```
function mytestfunction(){
    var arg1 = +prompt("give me a number",2);
    var arg2 = +prompt("give me another number",3)
    sum = arg1 + arg2;
    alert(sum)
}
```

Notice how you can address the issue either at the prompt or when you try to add them together.

3.2 – Passing arguments to a function

It becomes increasingly important throughout writing JavaScript to be able to pass arguments (or values) to your function. To do this is relatively simple in nature. The previous section mentioned parenthesized arguments; this is where you would put the variables names you want to reference in your function.

```
Example function definition
function addtogether(arg1, arg2){
    return arg1 + arg2;
}
```


The above example takes two arguments, adds them together, and returns the sum. To call the function and assign the return value to a variable named testResults would look like the following:

```
Example function call
var testResults = addtogether(5, 16);
```

The above function call would return 21 and set the testResults variable as the result.

```
Example function returns
testResults = 21
```

4.0 – DATA STRUCTURES: OBJECTS AND ARRAYS

Objects and Arrays are very similar in JavaScript. This section will explore some of the differences as well as ways to programmatically tell them apart from each other.

Some important facts about Objects and Arrays are:

.length property works on arrays, it doesn't work on objects

To loop through array values use:

```
for (var arrayCount = 0; arrayCount < myArray.length; arrayCount++){
    myArray[arrayCount] = "current array element";
}
```

To loop through object values use:

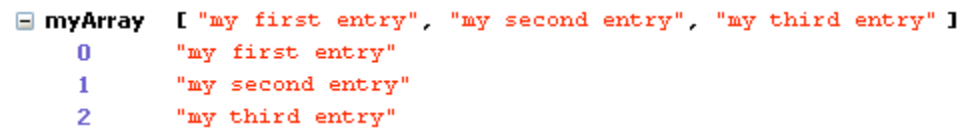
```
For (var objectProperty in myObject){
    myObject[objectProperty] = "current object property";
}
```

4.1 – Difference between objects and arrays

Arrays are numbered and have one level while an object can be multi-leveled and the elements are named instead of numbered.

```
Examples/Example_4-1_array.html
myArray = new Array();
myArray = ['my first entry', 'my second entry', 'my third entry'];
```

The above code will produce an array that looks like the following image when inspected with Firebug:



```
myArray [ "my first entry", "my second entry", "my third entry" ]
0      "my first entry"
1      "my second entry"
2      "my third entry"
```

```
Examples/Example_4-1_single-level_object.html
myObject = new Object();
myObject = {
    color: 'red',
    size: 'large',
    somethingelse: 'not me'
}
```

The above code will produce an object that looks like the following image when inspected with Firebug:

myObject	Object color=red size=large somethingelse=not me
color	"red"
size	"large"
somethingelse	"not me"

Examples/Example_4-1_multi-level_object.html

```
myObject = new Object();
myObject = {
  color: 'red',
  size: 'large',
  pages: {
    deleted: [1,5,21,34,55],
    added: [32,56,76],
    showadded: true,
    showdeleted: false
  },
  somethingelse: 'not me'
}
```

The above line of code will produce an object that looks like the following image when inspected with Firebug:

myObject	Object color=red size=large pages= Object
color	"red"
pages	Object deleted=[5] added=[3] showadded=true
added	[32, 56, 76]
0	32
1	56
2	76
deleted	[1, 5, 21, 2 more...]
0	1
1	5
2	21
3	34
4	55
showadded	true
showdeleted	false
size	"large"
somethingelse	"not me"

4.2 – An array can also be an object?

In JavaScript an Array is also an Object; but an Object is not necessarily an Array

Examples/Example_4-2_Array_Object.html

```
myArray = new Array();
myArray = ['my first entry', 'my second entry', 'my third entry'];
```

Clearly the above code defines an array named myArray and populates it with three values.

You can reference the values in an array like so:

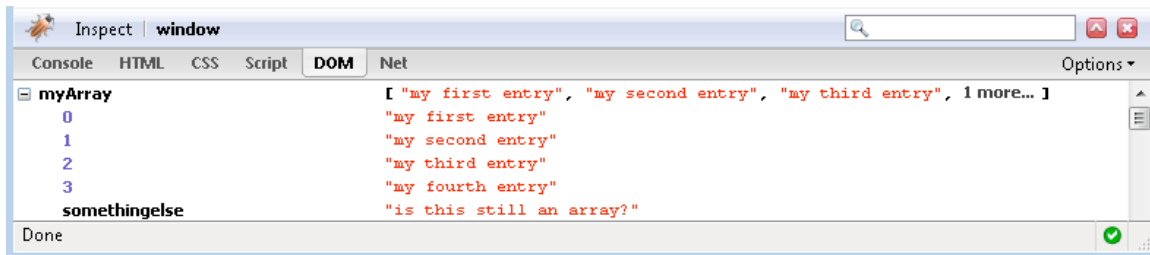
By number with brackets

```
myArray['3'] = 'my fourth entry';
```

NOTE: In JavaScript the numbering of array element's start at 0, so in the example above element number 3 is the fourth element...

You can also add a named value to an array. This is completely allowed in JavaScript, however it is not suggested because you will have both named and numbered elements in your array.

```
augmenting an array to also be an object  
myArray.somethingelse = "is this still an array?";
```



4.3 – Determining an object from an array

In JavaScript, an Array is an Object. The only difference is that it has un-named elements (that become numbered). We can determine if an object has un-named elements by using the instanceof keyword.

```
Examples/Example_4-3_object_or_array.html  
myObject = new Object();  
myObject = {  
    color: 'red',  
    size: 'large',  
    pages: {  
        deleted: [1,5,21,34,55],  
        added: [32,56,76],  
        showadded: true,  
        showdeleted: false  
    },  
    somethingelse: 'not me'  
}  
  
myArray = new Array();  
myArray = ["testing an array; item 1", "testing an array; item 2", "testing an  
array; item 3", "testing an array; item 4"]  
  
whatthearray = myObject instanceof Array;  
whattheobject = myObject instanceof Object;  
whatthetype = typeof myObject;  
  
whatthearray2 = myArray instanceof Array;  
whattheobject2 = myArray instanceof Object;  
whatthetype2 = typeof myArray;
```

The above script shows us that:

```
myObject instanceof Array = false  
myObject instanceof Object = true  
typeof myObject = object  
  
myArray instanceof Array = true  
myArray instanceof Object = true  
typeof myArray = object
```

This can be a little confusing because arrays also return true when tested with instanceof Object. So be sure to test for 'instanceof Array' to determine whether it is an array or an object.

Note: This may be a moot point since most developers will know whether or not their variable is an Object or an Array.

And when inspected with Firebug looks like:

```
myArray    [ "testing an array; item 1", "testing an array; item 2", "testing an array; item 3", 1 more... ]
0          "testing an array; item 1"
1          "testing an array; item 2"
2          "testing an array; item 3"
3          "testing an array; item 4"
myObject   Object color=red size=large pages=Object
color      "red"
pages      Object deleted=[5] added=[3] showadded=true
  added    [ 32, 56, 76 ]
    0      32
    1      56
    2      76
  deleted  [ 1, 5, 21, 2 more... ]
    0      1
    1      5
    2      21
    3      34
    4      55
  showadded true
  showdeleted false
  size      "large"
  somethingelse "not me"
whatthearray false
whatthearray2 true
whattheobject true
whattheobject2 true
whatthetype "object"
whatthetype2 "object"
```

5.0 - MODULARITY

Often times it becomes necessary, or convenient, to break up your program into modules. Breaking your program into modules can make it look less cluttered, easier to read, and often times easier to modify. To do this, you can save your JavaScript functions into .js files. Once your code is stored in .js files you can load them into the browser by placing a <script> tag with a src attribute into your HTML page; for example:

```
Examples/Example_7-0_Modularity.html
<HTML>
<HEAD>
<script type="text/javascript" src="js/ex_7-0_modularity.js"></script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

When writing JavaScript into its own .js file, you do not need to encapsulate it within a <script> tag. This is because when you load the .js file into the browser via the method above, the .js file is referenced within a <script> tag as the value of the SRC= attribute. You also do not need to encapsulate your JavaScript code within comments as previously shown. Lets look at the code in our js/myLibrary.js file:

```
Examples/js/ex_7-0_modularity.js
var test = 25;
function myAlert(x){
    // this comment is inside of my function
    alert("Your number plus 25 = " + x); // displays an alert
}

function getNum(){
    var c = +prompt('Quick, tell me a number!','');
    while(isNaN(c) || c==""){
        var c = +prompt('That wasn\'t a number... Tell me a number!','');
    }
    return c;
}
```

```
function myLoader(){
    myAlert(getNum()+test);
}

window.document.onload = myLoader();
```

5.1 – Order Matters

In JavaScript, the order in which you write your code matters. You cannot write code that depends on a function or object that has not yet been defined. Consider the following code:

```
<HTML>
<HEAD>
<script>
<!--
myLoader();
-->
</script>
<script type="text/javascript" src="js/ex_7-1_order_matters.js"></script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

In the example above, the myLoader function is defined within the 'js/ex_7-1_order_matters.js' file, but myLoader is called before the 'js/ex_7-1_order_matters.js' file has been loaded; this would generate an error:

```
Firebug Console Error:
myLoader is not defined
Examples/Example_7-1_Order_Matters.html
Line 5
```

Simply moving the calls that load the external JS file to above the myLoader() call would resolve that error:

```
<HTML>
<HEAD>
<script type="text/javascript" src="js/ex_7-1_order_matters.js"></script>
<script>
<!--
myLoader();
-->
</script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Another approach would be to place the myLoader() call in the body's onload event; that way the browser will not try to execute the myLoader() function until after the page is finished loading. This would look like:

```
<HTML>
<HEAD>
<script type="text/javascript" src="js/ex_7-1_order_matters.js"></script>
</HEAD>
<BODY onload="myLoader();">
</BODY>
</HTML>
```

The example script above will not execute the myLoader() function until the web page and all of its elements are finished loading.

6.0 – WEB PROGRAMMING

JavaScript can play an important role in web programming because there are a lot of things you can do with JavaScript that you cannot do with HTML alone. However this is not to say that JavaScript is limitless. There are actually quite a few limits to what JavaScript can do. For instance, a script cannot interact with the elements of a web page hosted on a different server other than that the script is hosted on. After all, it would be a little unsafe if the scripts running on myspace.com could interact with the DOM elements on bankofamerica.com.

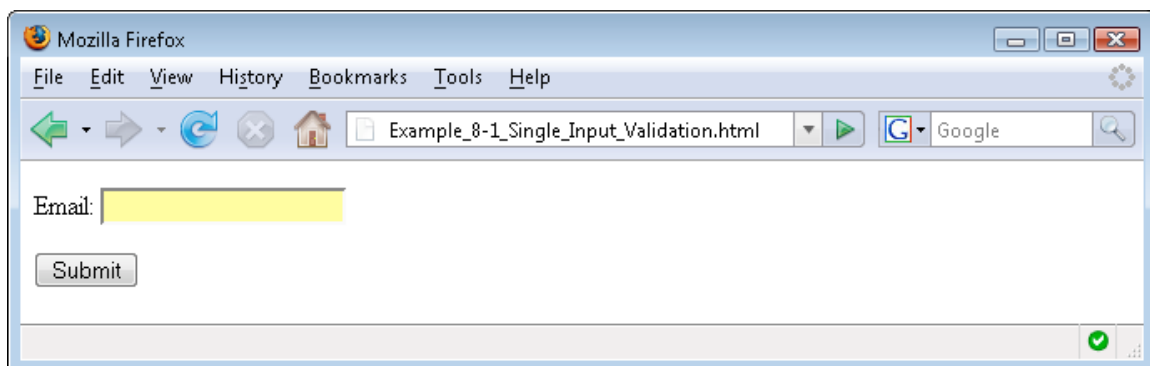
6.1 – Form Validation

Form Validation is one of the many uses of JavaScript on the web today. The concept of form validation is to check the form elements and prompt the user to correct any mistakes before submitting their contents to the server avoiding any page refreshes. Just like how the complexity of the form and the data you are requesting varies from task to task, the script that does the form validation will also vary in complexity. Let's look at a single input form; in this example we are asking for an email address.

```
Examples/Example_8-1_Single_Input_Validation.html
<HTML>
<HEAD>
<script type="text/javascript" language="JavaScript">
<!--
function validateForm(a) {
    var str = a.email.value;
    if((str.indexOf("@") > 0) && (str.indexOf(".") > 2) && (str != "")){
        return true;
    }else{
        str == "" ? alert("email field was left blank") : alert("\\" + str
+ "\" does not seem to be a valid email address");
        return false;
    }
}
-->
</script>

</HEAD>
<BODY>
<form action="#" method="POST" id="myform" name="myform" onsubmit="return
validateForm(this)">
    <p>Email: <input type="text" name="email" id="email"></p>
    <input type="submit" value="Submit">
</form>
</BODY>
</HTML>
```

The above sample code produces a very basic form that looks like the following image:



The HTML code behind this example form is very basic.

```
<form action="#" method="POST" id="myform" name="myform" onsubmit="return
validateForm(this)">
    <p>Email: <input type="text" name="email" id="email"></p>
    <input type="submit" value="Submit">
</form>
```

From the code snippet above we can see that when the form is submitted there will be a browser event “onSubmit” that calls the JavaScript code “return validateForm(this)”. **validateForm** is the name of our function, we pass along the keyword **this** so that our function can reference the context that called the function. By prepending the keyword **return** before our function call the form will not be submitted if the function returns false.

The JavaScript function **validateForm(a)** looks like this:

```
<script type="text/javascript" language="JavaScript">
<!--
function validateForm(a) {
    var str = a.email.value;
    if((str.indexOf("@") > 0) && (str.indexOf(".") > 2)){
        return true;
    }else{
        str == "" ? alert("email field was left blank") : alert("\"" + str
+ "\" does not seem to be a valid email address");
        return false;
    }
}
-->
</script>
```

When the form is submitted the function will be called. The function sets a local variable named **str** to the field identified by **ID=email** within our form that was passed to our function as **this**(now referenced as **a**):

```
var str = a.email.value;
```

The function then checks the value of the email field for a few specific items:

‘@’ is included and not the first character

‘.’ is included and is in the 3rd or higher character position (counting in JavaScript starts at 0)

The code snippet for this looks like:

```
if((str.indexOf("@") > 0) && (str.indexOf(".") > 2)){
    return true;
}
```

If both those conditions are met, the function returns **true** and the form is submitted. If the conditions are not met the following block of code is executed:

```
else{
    str == "" ? alert("email field was left blank") : alert("\"" + str
+ "\" does not seem to be a valid email address");
    return false;
}
}
```

At this point in the script we already know it is not a valid email, so the function simply tests if the value is empty, if it is empty the user is alerted with a message stating the “email field was left blank” otherwise they are alerted with a message stating that what they entered “does not seem to be a valid email address”. With either message the function also returns *false* telling the form not to be submitted.

Note: There is also a multi-input example of form validation with the examples for this course. Look for “Examples/Example_8-1_Form_Validation.html” in the examples folder.

6.2 – Image Rollovers

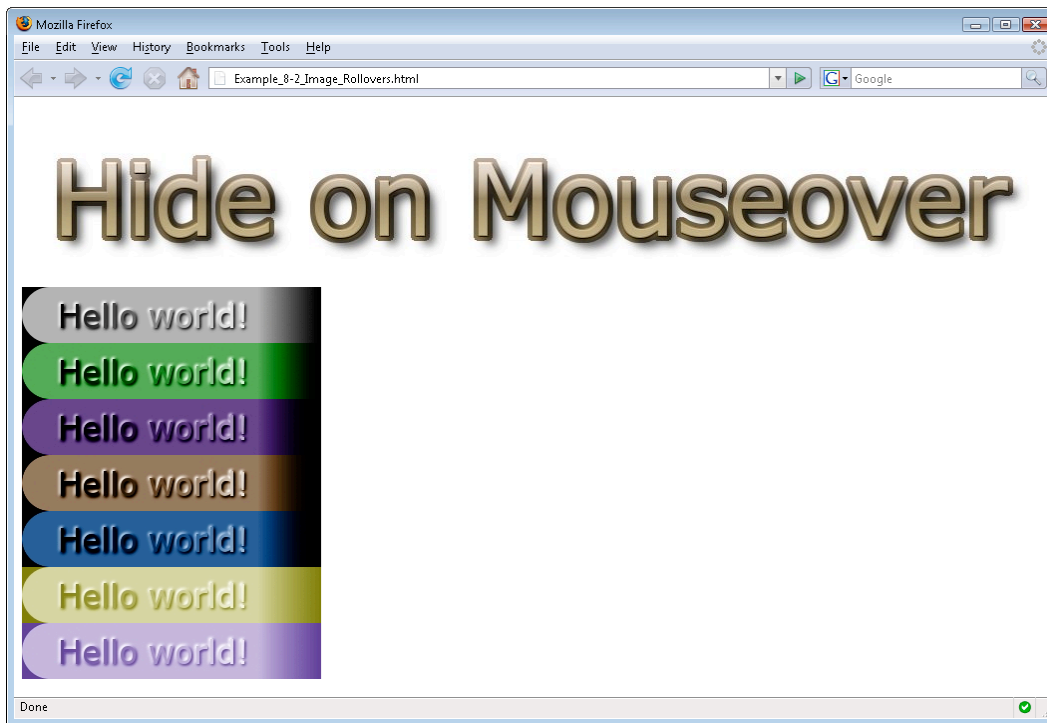
A certain level of pop can be achieved by adding roll over effects to images. In this subsection we look at changing elements based on the “onmouseover” browser event. Here is our example page:

```
Examples/Example_8-2_Image_Rollovers.html
<HTML>
<HEAD>
<script>
<!--
function hide(a){
    a.style.visibility = "hidden"; // set visibility to hidden
    a.onmouseout = function () {a.style.visibility = "";} // revert
visibility back to default
}

function changePic(a,b){
    var curSrc = a.src; // set variable curSrc to the current img src
    a.src = b; // set the current src to the
    a.onmouseout = function () {a.src = curSrc;} // set the onmouseout event
to revert back to the original
}

-->
</script>
</HEAD>
<BODY>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
</BODY>
</HTML>
```

When loaded in Firefox it looks like the following screenshot:



The first image on the web page has the following HTML:

```
<br>
```

We can see from this HTML tag that when the mouse is over this element the *hide* function will execute with *this* as its argument. Here is the *hide* function:

```
function hide(a) {
    a.style.visibility = "hidden"; // set visibility to hidden
    a.onmouseout = function () {a.style.visibility = "";} // revert
visibility back to default
}
```

From inspecting the function you can see that the first thing it does is set the argument passed in to the function as the variable 'a'. In the function call, the keyword *this* is passed as the argument, so the function's variable 'a' is going to be set to the element the function was attached to (i.e. 'a' is equal to our tag). The next thing the function does is set the style.visibility of the element to hidden. Finally the function sets the onmouseout event of the same element equal to an anonymous function that will reset the image's visibility back to default.

The remaining images on this webpage (all saying "Hello world!") have a different function attached to their "onmouseover" browser events. Here is the HTML for the remaining images:

```
<br>
<br>
<br>
<br>
<br>
<br>
```

```
<br>
```

As you can see, these images do not use the `hide()` function. These images use the `onmouseover="changePic(this, 'images/helloworld3.jpg');"` function. Here is the function definition:

```
function changePic(a,b){
    var curSrc = a.src; // set variable curSrc to the current img src
    a.src = b; // set the current src to the
    a.onmouseout = function () {a.src = curSrc;} // set the onmouseout event
to revert back to the original
}
```

As you can see from inspecting the code, this function expects two arguments; 'a' and 'b'. In the function calls that are attached to the image's onmouseover browser events the *this* keyword is passed as the first argument (set to 'a' in the function) and the relative path to the image we want visible while the mouse is over the image is passed as the second argument (set to 'b' in the function). The function first sets a variable *curSrc* equal to the current images source location. The function then sets the current source location of the image to the value of 'b' (the second argument passed in to this function). The last part of this function sets the "onmouseout" browser event of the image equal to an anonymous function that will revert the source location back to the original (saved as *curSrc*).

6.3 – Text Filling

JavaScript also has the ability to modify text on an already loaded web page. An example has been created to demonstrate this. Here is a look at the code for this example:

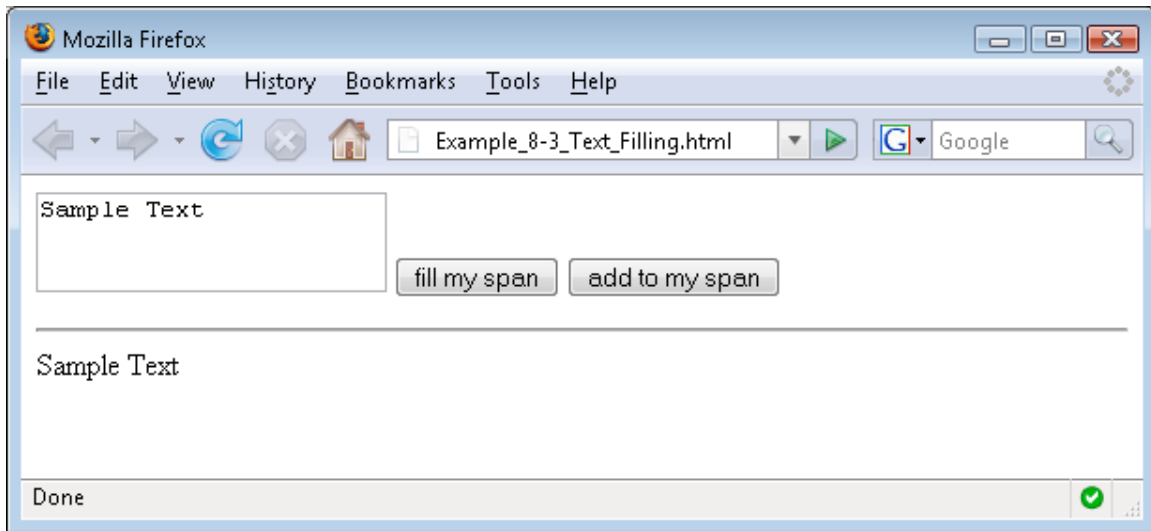
```
<HTML>
<HEAD>
<script>
<!--

function fillMySpan(){
    var x = document.getElementById("textToFill");
    var y = document.getElementById("fillHere");
    y.innerHTML = x.value;
}

function addToMySpan(){
    var x = document.getElementById("textToFill");
    var y = document.getElementById("fillHere");
    y.innerHTML += x.value;
}

-->
</script>
</HEAD>
<BODY>
<form ID="fillerForm">
<TEXTAREA ID="textToFill">Sample Text</TEXTAREA>
<input type="button" onclick="fillMySpan();" value="fill my span">
<input type="button" onclick="addToMySpan();" value="add to my span">
</form>
<hr>
<span ID="fillHere">Sample Text</span>
</BODY>
</HTML>
```

The above example produces an web page that looks like the following when rendered in Firefox:



The page consists of a simple form:

```
<form ID="fillerForm">
<TEXTAREA ID="textToFill">Sample Text</TEXTAREA>
<input type="button" onclick="fillMySpan();" value="fill my span">
<input type="button" onclick="addToMySpan();" value="add to my span">
</form>
```

And a simple ``:

```
<span ID="fillHere">Sample Text</span>
```

Separated by a horizontal rule:

```
<hr>
```

When the user clicks on the “fill my span” button the *fillMySpan()* function is executed. Here is the function definition for *fillMySpan()*:

```
function fillMySpan() {
    var x = document.getElementById("textToFill");
    var y = document.getElementById("fillHere");
    y.innerHTML = x.value;
}
```

What this function does is first set the variable ‘x’ to the elements identified by the *ID=textToFill* attribute. Then the ‘y’ variable is set to the element identified by the *ID=fillHere* attribute. The function then sets the *innerHTML* property of ‘y’ (i.e. our span identified by *ID=fillHere*) and sets it to the value property of ‘x’ (i.e. the text entered into the textarea form element identified by *ID=textToFill*). This will overwrite any text already visible in the span identified by *ID=fillHere*.

There is also a second button, labeled as “add to my span” which has a similar function assigned to it. When a user clicks on the “add to my span” button the *addToMySpan()* function is executed. Here is the function definition for the *addToMySpan()*:

```
function addToMySpan() {
    var x = document.getElementById("textToFill");
    var y = document.getElementById("fillHere");
    y.innerHTML += x.value;
```

```
}
```

As you can see, the `addToMySpan` is almost identical to the `fillMySpan()` function. The only difference is that we are appending the value property of the 'x' variable to the `innerHTML` property of the 'y' variable instead of resetting this property.

7.0 – BROWSER EVENTS

Web Apps written in JavaScript receive user and browser actions via browser events. Without them JavaScript would be a very uninteresting language. Browser events are used for a lot of things; for example, events are used during form validation, dynamic menus, etc. Browser events are a way for your scripts to react upon specific user actions.

7.1 – Standard Browser Events

The following is a list of standard browser events; more events do exist, but they are not fully supported by all browsers.

Category	Type	Attribute	Description
Mouse	Click	onclick	Triggers when the mouse or pointing device button is clicked over an element. A click is defined as a mousedown and mouseup over the same screen location. The sequence of these events are: mousedown mouseup click
	Double-click	ondblclick	Triggers when the mouse or pointing device button is double-clicked over an element
	Mousedown	onmousedown	Triggers when the mouse or pointing device button is pressed over an element
	Mouseup	onmouseup	Triggers when the mouse or pointing device button is released over an element
	Mouseover	onmouseover	Triggers when the mouse or pointing device is moved onto an element
	Mousemove	onmousemove	Triggers when the mouse or pointing device is moved while it is over an element
	Mouseout	onmouseout	Triggers when the mouse of pointing device is moved away from an element
Keyboard	Keypress	onkeypress	Triggers when a key on the keyboard is clicked. A keypress is defined as a keydown and a keyup on the same key. The sequence of these events are: keydown keyup keypress
	Keydown	onkeydown	Triggers when a key on the keyboard is pressed
	KeyUp	onkeyup	Triggers when a key on the keyboard is released

HTML Frame/Object	Load	onload	Triggers when the user agent finishes loading all content within a document, including window, frames, objects, and images. For elements, it triggers when the target element and all of its content has finished loading
	Unload	onunload	Triggers when the user agent removes all content from a window or frame. For elements, it triggers when the target element or any of its content has been removed
	Abort	onabort	Triggers when an object or image is stopped from loading before it is completely loaded
	Error	onerror	Triggers when an object/image/frame cannot be loaded properly
	Resize	onresize	Triggers when the document view is resized
	Scroll	onscroll	Triggers when the document view is scrolled
HTML Form	Select	onselect	Triggers when a user selects some text in a text field, including input and textarea
	Change	onchange	Triggers when a control loses the input focus and its value has been modified since gaining focus
	Submit	onsubmit	Triggers when a form is submitted
	Reset	onreset	Triggers when a form is reset
	Focus	onfocus	Triggers when an element receives focus either by the pointing device or by the keyboard
	Blur	onblur	Triggers when an element loses focus either by the pointing device or by the keyboard
User Interface	DOMFocusIn	ondomfocusin	Similar to the HTML focus event, but can be applied to any focusable element
	DOMFocusOut	ondomfocusout	Similar to the HTML blur event, but can be applied to any focusable element
	DOMActivate	ondomactivate	Triggers when an element is activated; for example, through a mouse click or keypress

Let's look at an example browser event. In this example I am using the same window.document.body.onload event that is used within the 4D Ajax Framework:

```
<body onload="dax_login('Guest',''); " onunload="dax_bridge.logout(); ">
```

Note: There are many resources online for browser events. A good place to start is http://en.wikipedia.org/wiki/DOM_Events

7.2 – Attaching Events to Elements

There are multiple ways to attach events to elements. This section will discuss the different methods and their drawbacks.

Inline Event Registration

In the early days of JavaScript, browsers only supported one type of event registration. This was known as inline event registration. In this type of event registration, event handlers were added as attributes to the HTML elements they were going to interact with. Consider the following:

Example

```

```

In the above example an alert is triggered when the user clicks on the image. You can also call a JavaScript function using inline event registration like so:

Example

```

```

The example above calls the function named *myFunction* when the user clicks the image.

Although inline event registration has been around the longest, and is pretty reliable, it still has one serious flaw. It requires you to write JavaScript code in line with the HTML, which is argued as a bad practice from the perspective of separating HTML presentation from JavaScript code.

Traditional Event Registration

Earlier we discussed how the window and document are objects. Within these objects are more objects and properties; including all of the elements within our webpage. Being that we can interact with each of these elements through JavaScript, it is possible to attach events to elements like so:

Example

```
element.onclick = doMyFunction();
```

The code snippet above attaches an event to the element named “element” that will trigger the ‘doMyFunction()’ function when “element” is clicked.

Let’s say you wanted to add an onload event to the body of a webpage, to trigger the ‘myStartup()’ function when the page is loaded. The JavaScript code to do this would look like:

Example

```
window.document.onload = myStartup();
```

The above JavaScript code snippet can be placed with the rest of your JavaScript code, as long as it is located outside of a function, and will be evaluated when it is sequentially loaded. If this code snippet was placed inside of a function, the event would only be registered once the function has been evaluated.

Advanced Event Registration

There may come a time when you need to attach multiple functions to the same event; although the event registration models do not directly support this, you can use anonymous functions to get

around this limitation. Let's say you wanted both `doMyFunction()` and `myStartup()` to be triggered when the page loads. This can be accomplished with the following code snippet:

Example

```
window.document.onload = function () {myStartup(); doMyFunction();}
```

The above code snippet is written on 1 line for simplicity, if your needs dictated, you could write the function on multiple lines like so:

Example

```
window.document.onload = function () {  
  
    /* This is my anonymous function that will be called  
       when the window finished loading */  
  
    myStartup(); // call the myStartup() function  
                // the myStartup function could check cookies,  
                browser, etc  
  
    doMyFunction(); // the doMyFunction function could do anything, it is  
                   used  
                    // in this example to demonstrate calling multiple  
                    // functions with one event.  
  
}
```

In the second example, many comments were added to demonstrate that an anonymous function can sometimes be easier to write because it allows more room for commenting and code structure.

8.0 – QUESTIONS

There will be time at the end of the session for questions.