



Browser Scripting

Release: 2010-07-20

Table Of Contents

- [ECMAScript / JavaScript](#)
 - ◆ [Comments](#)
 - ◆ [Data Types](#)
 - ◆ [Operators](#)
 - ◆ [Variables](#)
 - ◆ [Constants](#)
 - ◆ [Arrays](#)
 - ◆ [Functions](#)
 - ◆ [Objects](#)
 - ◆ [Flow Control](#)
 - ◆ [Error Handling](#)
 - ◆ [Regular Expressions](#)
- [Webpages](#)

ECMAScript / JavaScript

ECMAScript is the core standard of object-orientated script languages usually provided for client-side Web Application functionality such as in web browsers. JavaScript was invented by Netscape and continued by Mozilla and is built on top of ECMAScript (Edition 2) since version 1.3. JavaScript's syntax may be similar to Oracle's Java programming language but JavaScript is not a scripted version of Java – it is a completely separate entity. Microsoft's JScript implements some of JavaScript and is also based on some of ECMAScript Edition 2 since version 2.

Languages based on ECMAScript are object-orientated, encapsulating data types like objects, strings, numbers, arrays, dates, errors and functions. The language is also case sensitive: it cares whether the keywords and variable names are lowercase, UPPERCASE, camelCase or generally miXeD cAsE. In such environments that render webpages the scripts will also have access to objects that expose parts of the webpage.

Comments

Comments are provided in two types: Multiple Line Comments:

```
/* This is an ECMAScript, JavaScript  
and JScript multi line comment. */
```

Figure 1: Multiple line comment example

Single Line Comments start with two slashes and end at the current line's end or the end of the script:

```
// This is a scripting single line comment.
```

Figure 2: Single line comment example

Data Types

With most programming languages that you compile to produce the application such as C, C++ and Java, you need to specifically state the type of the data (Static Typing). ECMAScript based languages, which were interpreted languages, automatically determine what the type of data is (Dynamic Typing).

Currently most ECMAScript and JavaScript implementations are shifting from script engines to script virtual machines, which compile the script at runtime to improve processing and response performance but still supports Dynamic Typing.

Strings are one or more characters such as letters, number digits and punctuation together and surrounded by either matching double quotes, "This is a string.", or single quotes, 'This is a string.'. The double quote is the preferred

Browser Scripting - Legend Scrolls

choice by many scripters. A backslash (\) is a special character within strings. Within a double quoted string you can have a double quote as part of the string data by preceding the double quote with the backslash as "She said \"Hello!\" to the world". Otherwise the string would end before you intended. Similarly for apostrophes in single quoted strings as 'It doesn\'t shine anymore'. To break up lines within any quoted string you use the newline character (\n) as "The sun glows.\nRain is wet."

Numbers or digits are either whole numbers (or integers), one or more zero to nine characters, optionally with a minus sign before it such as 32 or -68. Or decimal numbers (sometimes called float, double or real numbers) such as 32.183, -68.78802, 1.193e15, -1.03e2, -1.04e-3 or 2.22e-3.

Boolean values are something is (true) or something is not (false). A native boolean value will either be the literal word true without quotes or the literal word false without quotes. In a condition such as if something is this or something is not that, an empty string ("") or (") would be equivalent to the boolean false as would the number zero (0) or (0.0), otherwise they would be equivalent to the boolean true.

Operators

Operators are one or two or the odd three character thing that performs a basic function between one or two values.

Object Property or Method Pointer (dot .):

Used in object syntax between object name and either property name or method name.

Standard Assignment (single equals =):

Assigns a string, number, array, expression, object or other to a variable.

String Concatenation (plus sign +):

Connects at least one string with another string or variable or number or other.

Math Addition (plus sign +):

Sums or adds two numbers together, adds the number from the right of the plus sign to the number on the left of the plus sign.

Math Subtraction (minus sign or dash -):

Subtracts or deletes one number from another number, deletes the number from the right of the minus sign from the number on the left of the minus sign.

Math Multiplication (asterisk or star *):

Browser Scripting - Legend Scrolls

Multiply or times two numbers together, times the number on the left of the asterisk with the number on the right of the asterisk.

Division (forward slash /):

Divides or shares one number from another number, divides the number from the left of the divide sign with the number on the right of the divide sign. If the result is not a whole number than it provides the remainder as a decimal such as `12 / 5` is `2.4`.

Modulus (percentage sign %):

Divides or shares one number from another number, divides the number from the left of the modulus sign with the number on the right of the modulus sign. If the result is not a whole number then it does not provide the remainder such as `12 % 5` is `2`.

Increment (two plus signs ++):

A shorthand to something equals something plus 1. Instead of `result = result + 1` you do `result++` which provides the current `result` and then adds 1. Or you can do `++result` which adds 1 and then provides the updated `result`.

Decrement (two dashes or minus signs --):

A shorthand to something equals something minus 1. Instead of `result = result - 1` you do `result--` which provides the current `result` and then deletes 1. Or you can do `--result` which deletes 1 and then provides the updated `result`.

Equals (two equal signs ==):

Compares the value of a variable or literal value on the left of the 'equals' with the value of the variable or literal value on the right of the 'equals'. If they are the same then it returns `true` otherwise returns `false`.

Not Equals (exclamation mark and single equals !=):

Compares the value of a variable or literal value on the left of the not equals with the value of the variable or literal value on the right of the not equals. If they are not the same then it returns `true` otherwise returns `false`.

Strictly Equals (three equal signs ===):

Strictly compares the type and value of a variable or literal value on the left of the strict equals with the type and value of the variable or literal value on the right of the strict equals. If both the type and value are the same then it returns `true` otherwise either are different then returns `false`.

Strictly Not Equals (exclamation mark and two equal signs !==):

Strictly compares the type and value of a variable or literal value on the left of the strict not equals with the type and value of the variable or

Browser Scripting - Legend Scrolls

literal value on the right of the strict not equals. If either the type or value are different then it returns **true** otherwise if both are the same then returns **false**.

Less Than (left pointy bracket <):

Compares if the left value is less than the value on the right. If so, returns **true**, otherwise returns **false**.

Less Than And Equal To (left pointy bracket and equals <=):

Compares if the left value is less than or equal to the value on the right. If so, returns **true**, otherwise returns **false**.

Greater Than (right pointy bracket >):

Compares if the left value is greater than the value on the right. If so, returns **true**, otherwise returns **false**.

Greater Than And Equal To (right pointy bracket and equals >=):

Compares if the left value is greater than or equal to the value on the right. If so, returns **true**, otherwise returns **false**.

Logical NOT (exclamation mark !):

Used in conditions. States the value directly to the right is considered as a negative. If the value returns (boolean) **false** then the condition is **true** and the instructions are processed.

Logical AND (two ampersands &&):

Used in conditions. Checks both the value to the left and to the right. If both values return (boolean) **true** then the overall result is **true** and the instructions are processed.

Logical OR (two vertical bars ||):

Used in conditions. Checks both the value to the left and to the right. If either value returns (boolean) **true** then the overall result is **true** and the instructions are processed.

Additive Assignment (plus sign and equals +=):

Equivalent to something = something + someOtherThing for numbers (**answer += 3**; if **answer** was **2** it would now be **5**) and also appending strings to existing strings (**paragraph += "An extra line."**);).

Subtractive Assignment (a dash and equals -=):

Equivalent to something = something - someOtherThing (**answer -= 3**; if **answer** was **8** it would now be **5**).

Multiplication Assignment (an asterisk and equals *=):

Equivalent to something = something * someOtherThing (**answer *= 3**; if **answer** was **3** it would now be **9**).

Browser Scripting - Legend Scrolls

Division Assignment (a forward slash and equals /=):

Equivalent to something = something / someOtherThing (answer /= 5; if answer was 12 it would now be 2.4).

Modulus Assignment (a percentage sign and equals %=):

Equivalent to something = something % someOtherThing (answer %= 5; if answer was 12 it would now be 2).

Variables

To store information such as literal numbers, objects, dates or text strings you use variables like you would put a letter in an envelope or a product into a box.

An empty variable will have the special type `undefined`. You can create an empty variable and assign a value later or create the variable with the value (initialise the variable). The `var` keyword can be used to formally declare the variable but this keyword is optional.

```
// Create an empty variable  
var myBox;
```

```
// Creating a group of empty variables  
var myBox, startNumber;
```

```
// Assigning a value to the empty variable  
myBox = "The blue box";
```

```
// Create a variable containing a string  
var myBox = "This is a string.";
```

```
// Create a variable containing a whole number (an integer)  
startNumber = 38;
```

```
// Create a variable containing a decimal number (a real number)  
startNumber = 1.3;
```

Figure 3: Creating variables

Variable names must start with either a lower or upper case letter or an underscore (`_`) or a dollar sign (`$`), then any number of underscores, dollar signs, digits (`0` to `9`), lower or upper case letters. ECMAScript is case-sensitive and so lower and upper case letters are different. Such as the variable name `myVar` is not the same as `myvar`, `MYVAR`, `Myvar`, `myVAR`, etc. These could be five separate variables.

These simple variables are known as scalar variables. As you may have noticed, at the end of each statement or expression a semi-colon is used to indicate the end of the scripting statement.

If you want to create and initialise a variable but don't want to have an actual

Browser Scripting - Legend Scrolls

value yet, you can use the special `null` keyword as the value (no quotes around the value) as `var myMessage = null;`.

Simple mathematical expressions can be simply coded as assigning the answer into the variable but instead of a simple number or string, you state a math expression:

```
var total = 23 + 38 + 187;
var status = 5.8 - 1.3;
var per = 62 * 3;
var share = 102 / 2;
var result = ((48.3 + 24.2) * 3) / 0.5
var ultraTotal = total + share + per - status - result;
```

Figure 4: Simple Math

Making the variable `total` contain the number `248`.

The variable `status` would contain the decimal number `4.5`.

Variable `per` would have the number `186`.

For `share` has `51`.

Also `result` is `435`. As in Math you can surround parts of an expression with parenthesis (normal brackets) to control which parts are calculated before others.

And you can re-use variables within the expression making `ultraTotal` the value of `-47.5`. So in order to use the variable you simply state the variable name.

Multiple strings, variables and even numbers to be in a single string can be concatenated (connected together) by using the plus sign (+).

```
var message = "The total of the calculation is " + result + " units.";
```

Figure 5: Concatenating into a single string.

So if the result of the variable `result` is `435` then the value of `message` is 'The total of the calculation is 435 units.'

Constants

Constants are like variables but instead of the optional `var` keyword you use the mandatory `const` keyword and you must define a value that is not the `null` keyword. A constant's name must only start with a lower or upper case letter or an underscore followed by any number of underscores, digits or lower or upper case letters. The value of a constant cannot change after its creation.

```
// Create a couple of constants
const coreValue = 23.33;
const companyName = "MyMetaCorp";
```

```
// Use the constants
var totalValue = coreValue + 46.8;
```

```
var myMessage = "The total value for " + companyName + " is " + totalValue + ".";
```

Figure 6: Creating and using constants

Arrays

Arrays are variables with multiple values in one. Using square brackets, [], and a whole number as a zero based index (first index is 0, then 1, etc.) allocates multiple values to a single array variable.

```
var fruit[0] = "orange";  
fruit[1] = "apple";  
fruit[2] = "grape";  
fruit[3] = "banana";  
fruit[4] = "pear";
```

Figure 7: A simple index array.

If you just stated the variable `fruit`, the value would be 'Array'. In order to access a particular element of the array you would state the array name and it's index as `fruit[3]` to get the value of `banana`.

An associative array uses a name instead of an index:

```
var fruit["first"] = "orange";  
fruit["another"] = "apple";  
fruit["more"] = "grape";  
fruit["thisone"] = "banana";  
fruit["thelast"] = "pear";
```

Figure 8: A simple associative array.

And so to refer to `banana` you would state `fruit["thisone"]`.

An array literal uses square brackets to state the comma separated list of elements and assign the whole thing to a variable. This would act like an index array.

```
var fruit = ["orange", "apple", "grape", "banana", "pear"];
```

Figure 9: An array literal

Functions

You can encapsulate or noticeably group statements and expressions into a subroutine or function. Using the keyword `function` and then giving the function a name followed by parenthesis, which may contain parameters, and then braces (curly brackets { }) for the function body. The statements and expressions go within the braces.

You can feed the function with one or more pieces of information by either having a name or a comma separated list of names within the parenthesis. These names are the parameters of the function. Then you can use the parameters as variables within the function body.

Browser Scripting - Legend Scrolls

```
function myFunc () {  
    var calc = (186 * 3) / 2;  
    var term = calc + 38;  
    var total = 18.46 + term;  
    return total;  
} // (function) myFunc
```

Figure 10: Creating a simple function.

The `return` keyword provides the output of the function.

```
function myDynFunc (temp, degrees) {  
    var calc = (186 * 3) / temp;  
    var term = calc + degrees;  
    var total = 18.46 + term;  
    return total;  
} // (function) myDynFunc
```

Figure 11: Creating a simple function with parameters.

The comments after the end brace, `}`, is not essential for the creation of the function but I have found them very useful especially for functions with loads of lines of statements and expressions and so you don't have to scroll and look back up and find the start of the function to see what the end brace is ending. The comment will tell you right there. After creating the function you use the function by assigning a variable with the function. The variable will then have the result of the function.

```
calcTotal = myFunc();  
calcDynTotal = myDynFunc(2, 38);  
calcDynTotal2 = myDynFunc(3, 18);
```

Figure 12: Using functions

Resulting in both `calcTotal` and `calcDynTotal` would each provide `335.46`.

And `calcDynTotal2` would provide `222.46`.

Variables that are created within functions only exist within that particular function as a local variable. Just as variables created in the general flow of the script are global variables and are available to the general script. So you can have two variables with the same name but one is created and used generally and the other is created and used only within a particular function. They have different variable scopes.

A collection of core functions that can be used include `isFinite()` which takes a number as its parameter. It returns `false` if the parameter is not a number or the positive or negative infinity value; otherwise returns `true`.

`isNaN()` with a parameter returns `true` if the parameter is not a number and `false` if is.

`parseInt()` tries to convert a string parameter into an integer. An optional second parameter states which base to convert into such as `8` (base 8) is octal (0-7), `10` (base 10) is decimal (0-9) and `16` (base 16) is hexadecimal (0-9 a-z/A-Z). Base 10 is the default if the second parameter is not used. If part of

Browser Scripting - Legend Scrolls

the string can't be converted according to the second parameter then that part till the end of the string will be ignored. If none of the string can be converted then `NaN` (not-a-number) will be returned.

`parseFloat()` tries to convert a string parameter into a floating point or decimal number. If part of the string can't be converted then that part till the end of the string will be ignored. If none of the string can be converted then `NaN` (not-a-number) will be returned.

The `String()` and `Number()` functions take a single parameter that can be any type of object (see next section) and convert it into a string object via `String()` or into a number object via `Number()`.

Objects

Another way of variables with multiple pieces of related information are objects. Unlike compiled languages, objects in ECMAScript are prototype-based: variables that you add simple scalar variables or arrays as properties of the object and functions that you add as methods of the object, all at runtime.

For instance assume we have created a product object with `name`, `mainColour` and `dimensions` properties:

```
myProduct1.name = "truck";
myProduct1.mainColour = "blue";
myProduct1.dimensions["width"] = 108;
myProduct1.dimensions["height"] = 68;
```

Figure 13: An object with properties

Creating an object can be done in one of two ways either by creating the structure of the object via a function using the `this` keyword and then assigning the special function to a variable as an object using the `new` keyword:

```
function Product (name, mainColour, dimensions) {
    this.name = name;
    this.mainColour = mainColour;
    this.dimensions = dimensions;
} // (Object Definition) Product
```

```
var prodDim["width"] = 108;
prodDim["height"] = 68;
```

```
var myProduct1 = new Product("truck", "blue", prodDim);
```

Figure 14: Creating an object via a function.

Or since JavaScript 1.2 and ECMAScript Edition 2 you can use object initialisers using braces and property names as names, numbers or strings and their values:

```
var myProduct1 = {
    name: "truck",
```

Browser Scripting - Legend Scrolls

```
    mainColour: "red",
    dimensions: [108, 68]
} // (Object) myProduct1
```

Figure 15: Creating an object via object initialisation.

After creating an object you can add an extra property simply for instance `myProduct1.group = "model"`; but this would only add this property and value specifically to the `myProduct1` object and not any other objects based on `Product`.

To add a new property to a pre created object definition you use the `prototype` property:

```
Product.prototype.group = null;
myProduct1.group = "model";
```

Figure 16: Adding more properties to pre created object definitions

This adds the `group` property to all objects based on `Product`.

To delete properties from object instances such as `myProduct1.group` you use the `delete` keyword as `delete myProduct1.group` would delete the `group` property. Also `delete myProduct1` would delete the whole object if the object variable was not created with `var`.

Adding methods to an object is done by creating a function as usual and the same way you add a property to the object but using the function name as the value of the property making it a method.

```
function objLabel () {
    return "This product is a " + this.name + " and it's main colour is " +
this.mainColour + ". The product's dimensions are " + this.dimensions["width"] + " by " +
this.dimensions["height"] + ".";
} // (Method) objLabel
```

```
function Product (name, mainColour, dimensions) {
    this.name = name;
    this.mainColour = mainColour;
    this.dimensions = dimensions;
    this.toLabel = objLabel;
} // (Object Definition) Product
```

Figure 17: Adding an object method

Then you can use `myProduct1.toLabel()`; which will return "This product is a truck and it's main colour is blue. The product's dimensions are 108 by 68."

There is a collection of predefined core objects available to use. Some of these include:

String Objects

To create a string object you use the `new` keyword as `var s1 = new String("The BluE box");`.

Browser Scripting - Legend Scrolls

String objects have a property called `length` that provides how many characters are contained in the string. `s1.length`; would provide the number `12` as there are 12 characters including spaces in the string.

Two of the string methods are `toUpperCase()` and `toLowerCase()`. Such as `s1.toLowerCase()`; would provide "the blue box".

Providing a part of the string is done using the `substring()` method. There are 2 parameters for this method: the first is the zero-based index of the character starting the string part such as the first character in the string is zero, the second is 1. The optional second parameter is the index of the character ending the string part; if higher than the first parameter, then the substring is from the first parameter's character and towards the right of the string. If the second parameter is lower than the first then it is from the first parameter's character and towards the left of the string.

`s1.substring(0, 1)`; returns "T".

`s1.substring(1, 5)`; returns "he B".

`s1.substring(6, 0)`; returns "The Bl".

Also the `substr()` method is pretty much the same.

You can split strings with the `split()` method with the separator string as the first parameter and an optional number as the second parameter. The separator string is found in the main string and separates the substrings before and after each instance of the separator string; the second parameter is how many instances of the separator string is searched for. An array is the result of the `split()` method.

Joining string objects together can be accomplished by the `concat()` method with a comma separated list of string objects.

The `replace()` method allows you to do search and replace abilities such as replace 'tin' with 'tan':

```
var txt = new String("This tin has changed colour");  
var txt2 = txt.replace("tin", "tan");
```

Figure 18: search and replace with String

Number Objects

The Number Object has some constants that you use literally such as:

`Number.MIN_VALUE` which provides the minimum value possible;

`Number.MAX_VALUE` provides the highest value possible;

`Number.NaN` provides a special Not-A-Number value;

`Number.POSITIVE_INFINITY` provides the value that represents Infinity at the positive perspective;

Browser Scripting - Legend Scrolls

`Number.NEGATIVE_INFINITY` provides the value that represents Infinity at the negative perspective.

Objects created from the Number Object or a number literal in a variable can use some Number methods.

```
var num1 = 28;
var num2 = new Number(489);
```

Figure 19: Creating more numbers

Using the `toString()` method returns the number as a string in the typical decimal form by default. You can provide a number between `2` and `36` as the parameter to change the output format such as `2` for base 2 (binary), `8` for octal (0-7), `10` for decimal (0-9) or even `16` for hexadecimal (0-9 a-z / A-Z).

To control how much precision is used when returning a decimal number you use the `toFixed()` method. With a number from `1` to `21` by standard (some implementations may support higher precisions) as the parameter will return the decimal to that number of significant digits.

```
// Returns 3.9283;
var num3 = 3.92828739278;
num3.toFixed(5);
```

```
// Returns 39.283;
var num3 = 39.2828739278;
num3.toFixed(5);
```

Figure 20: Number precision

Without a parameter the `toFixed()` method just returns the exact number.

Array Objects

Arrays can be created via the Array Object, such as:

```
var fruit = new Array("orange", "apple", "grape", "banana", "pear");
var fruit2 = new Array(5);
```

Figure 21: Array objects

The second example creates an array with 5 empty array elements. Populating the empty array objects is done the same way you populate normal arrays.

A main array property is the `length` such as `fruit.length`; will provide the number of array elements in the array.

Joining multiple new array elements with an existing array you have a comma separated list of array elements as the parameters of the `concat()` method. The `join()` method uses a 'joiner' string parameter and joins each array element together with the 'joiner' string between each element and the results in a simple string. If the 'joiner' is left empty then the comma (,) character is used.

Also a method for reversing the contents of an array just by using the `reverse()`

method. Ascending sort for an array contents is done via the `sort()` method.

Date Objects

Handling dates and times is done via the Date Object.

```
var now = new Date();  
var then1 = new Date("August 25, 2008 14:18:02");  
var then2 = new Date(2008, 08, 25, 14, 18, 02);  
var then3 = new Date(2008, 08, 25);
```

Figure 22: Date Objects

Without any parameters, the date object selects the current date and time. With a full month name followed by a space and the day number, a comma, the full year number, a space and the time in hours, a colon, minutes, colon and seconds (a UTC Time) will select that date and time and put that into the date object variable. Alternatively you can have comma separated parameters of the year, month, day and optional hours, minutes and seconds all as numbers.

Date methods include get based methods that have no parameters and set based methods that have a single parameter of the particular time piece to set:

- `getDate()` / `setDate()` returns / sets the local day of the month (1 to 31),
- `getDay()` returns the local day of the week (0 for Sunday to 6 for Saturday) (there is no `setDay` as this is done automatically),
- `getFullYear()` / `setFullYear()` returns / sets the local 4 digit year,
- `getMonth()` / `setMonth()` returns / sets the local month (0 for January to 11 for December),
- `getHours()` / `setHours()` returns / sets the local hour (0 to 23 (24 hour clock)),
- `getMinutes()` / `setMinutes()` returns / sets the local minutes (0 to 59),
- `getSeconds()` / `setSeconds()` returns / sets local seconds (0 to 59) and
- `getMilliseconds()` / `setMilliseconds()` returns / sets local milliseconds (0 to 999).

There are Coordinated Universal Time (UTC) versions of the above date methods as `getUTCDate()`, `setUTCDate()`, `getUTCDay()`, `getUTCFullYear()`, `setUTCFullYear()`, `getUTCMonth()`, `setUTCMonth()`, `getUTCHours()`, `setUTCHours()`, `getUTCMinutes()`, `setUTCMinutes()`, `getUTCSeconds()`, `setUTCSeconds()`, `getUTCMilliseconds()` and `setUTCMilliseconds()`.

Also `getTime()` / `setTime()` returns / sets the date and time as a number representing milliseconds since midnight, January 1st, 1970 UTC (Computer Epoch) and `getTimezoneOffset()` returns the current locale's timezone in minutes.

Browser Scripting - Legend Scrolls

A couple of obsolete methods: `getYear()` and `setYear()` returns and sets the year only with 2 digits. These two are still supported for backwards compatibility but should not be used any more. Use the full year and UTC full year get and set methods instead.

Math Objects

The Math object is used literally such as `Math.PI` rather than using `new` and putting into a variable. This object provides typical math constants as properties such as:

Euler's Constant (`Math.E`),

Natural Logarithm of 2 (`Math.LN2`) or 10 (`Math.LN10`),

Logarithm Base 2 (`Math.LOG2E`) or 10 (`Math.LOG10E`),

PI (`Math.PI`),

Square root of 2 (`Math.SQRT2`) and 1 over the square root of 2 (`Math.SQRT1_2`).

Plus typical math functions as methods such as:

`Math.abs()` takes a number and provides the absolute number of it such as `Math.abs('-5')`; provides 5;

`Math.sin()`, `Math.cos()`, `Math.tan()` each take a number (in radians) and provide either the sine, cosine or tangent respectively;

`Math.asin()`, `Math.acos()`, `Math.atan()` each take a number between -1.0 and 1.0 and provide either the arcsine, arccosine or arctangent respectively (in radians);

`Math.atan2()` takes a y-coordinate as the first parameter and an x-coordinate as the second parameter and provides a value between -Pi and Pi in radians;

`Math.exp()` takes a number as the exponent for Euler's Constant;

`Math.log()` takes a number and returns the Natural Logarithm of that number;

`Math.round()` takes a number and rounds it to the nearest integer: if .5 and higher decimal then rounds up or below .5 decimal then rounds down;

`Math.ceil()` for ceiling provides the smallest integer greater than or equal to the number parameter (in other words always rounds up);

`Math.floor()` provides the highest integer less than or equal to the number parameter (in other words always rounds down);

`Math.min()` provides the smallest number out of the comma separated list of number parameters;

`Math.max()` provides the highest number out of the comma separated list of number parameters;

Browser Scripting - Legend Scrolls

`Math.pow()` takes a **base** number as the first parameter and an **exponent** or power number as the second parameter and returns the result of **base** to the power of **exponent**;

`Math.random()` has no parameters, just returns a random number from 0.0 to below 1.0 and is seeded by the current datetime;

`Math.sqrt()` provides the square root of the number parameter.

Flow Control

Flow Control includes loops such as `for`, `while` and conditions such as `if...else` and `switch`.

Looping with `for` allows you to sequence through a known range such as from 1 to 200. You first set a variable (usually `i` or `j`) with the starting number, set the same variable to be less than (`<`) the highest number plus one followed by the same variable to be incremented (`++`). Or instead of less than you may use less than or equal to (`<=`) the highest number. Also decrements can be set too (`--`).

```
for (i=1; i<201; i++) {  
    // Set commands to process for each incrementation.  
}  
for (i=1; i<=200; i++) {  
    // Set commands to process for each incrementation.  
}  
for (i=200; i>0; i--) {  
    // Set commands to process for each decrementation.  
}  
for (i=200; i>=1; i--) {  
    // Set commands to process for each decrementation.  
}
```

Figure 23: Looping with `for`

Looping with `while` allows the instructions to action only during a condition's lifetime:

```
while (flipSwitch == "on") {  
    // Do this  
}  
while (flipSwitch != "off") {  
    // Do this  
}
```

Figure 24: A `while` loop

Also a condition as 'if this, otherwise that' can be done for instance:

```
if (setting=="level 2") {  
    // Do this  
} else {
```

Browser Scripting - Legend Scrolls

```
    // Do that  
}
```

Figure 25: If...else condition.

Multiple conditions:

```
if (setting=="level 2") {  
    // Do this  
} else if (group=="lower") {  
    // Do this instead  
} else if (flipSwitch=="on") {  
    // Or do this  
} else {  
    // Or do that  
}
```

Figure 26: If...else..if condition.

A `switch` is like an `if...else...if` with the variable having the same name in each condition for instance an `if...else...if` such as the following can also be a `switch` statement as the following:

```
// The if...else...if  
if (setting=="level 2") {  
    // Do this  
} else if (setting=="level 5") {  
    // Do this instead  
} else if (setting=="level 28") {  
    // Or do this  
} else {  
    // Or do that  
}
```

```
// The switch  
switch (setting) {  
    case "level 2":  
        // Do this  
        break;  
    case "level 5":  
        // Do this instead  
        break;  
    case "level 28":  
        // Or do this  
        break;  
    default:  
        // Or do that  
}
```

Figure 27: Switch condition.

The value of each `case` is usually a number or a string. The `break` keyword is used to break out of the loop or condition. As in the case of `switch` (pun half intended): if the first `case` did not have the `break` keyword then if `setting` was

Browser Scripting - Legend Scrolls

'level 2' then it would process the commands for both 'level 2' and 'level 5' but not 'level 28' as the `break` keyword stops the `switch` before that point. If none of the cases match then the commands for the `default` will process.

The `default` part of a `switch` is optional. If the `default` part is missing then the last `case` does not need a `break` keyword as this is the end of the `switch` now.

Some special control flow for objects include `for...in` to sequence through each object property. Providing a variable to store each property for the current iteration (or revolution) such as `i` or `prop` and the actual object name:

```
var props;
for (i in myProduct1) {
    props += i + ": " + myProduct1[i] + ", ";
}

```

Figure 28: Looping through objects with `for...in`

Error Handling

Object orientated languages have special kinds of errors that could be produced in a program or script such as the wrong type of data was submitted to a function, no data was submitted or some other condition that the script was not able to deal with. These types of errors are called exceptions.

If you need a part of the script to specifically generate an exception because the function requires a specific data type or requires only a specific range of values for instance; then you can generate an exception using the `throw` keyword and a string or number, variable or even an object.

```
var value;
if (input<10) {
    value = input;
} else {
    throw "invalidRangeException";
}

```

Figure 29: Throwing an exception

Exceptional error handling or exception handling can be done using the `try...catch...finally` statements. The `try` part contains the commands that might get errors. To provide the response of the exceptions you use the `catch` part. And in the optional `finally` part are any commands or statements that can be processed regardless of the outcome of the commands in the `try` part.

```
var value, response;
try {
    if (input<10) {
        value = input;
        response = value;
    } else {
        throw "invalidRangeException";
    }
}

```

Browser Scripting - Legend Scrolls

```
    }  
  } catch (e) {  
    response = "Please provide a number below 10.";  
  }  
}
```

Figure 30: A try...catch example

```
var value, response;  
try {  
  if (input<10) {  
    value = input;  
    response = value;  
  } else {  
    throw "invalidRangeException";  
  }  
} catch (e) {  
  response = "Please provide a number below 10.";  
} finally {  
  response += " Enter a new value?";  
}
```

Figure 31: A try...catch...finally example

If there is more than one exception that could occur then you may use more than one `catch` part, each catching a particular exception. Optionally you can also have a `catch` to catch any exceptions that do not match the others (like the default part of a `switch`):

```
var value, response;  
try {  
  if (input<2) {  
    throw "valueIsTooLowException";  
  } else if (input>8) {  
    throw "valueIsTooHighException";  
  } else {  
    value = input;  
    response = value;  
  }  
} catch (e if e == "valueIsTooLowException") {  
  response = "Input is too low, please enter a number above 1.";  
} catch (e if e == "valueIsTooHighException";) {  
  response = "Input is too high, please enter a number below 9.";  
} finally {  
  response += " Enter a new value?";  
}
```

Figure 32: A multiple catches example

```
var value, response;  
try {  
  if (input<2) {
```

Browser Scripting - Legend Scrolls

```
        throw "valueIsTooLowException";
    } else if (input>8) {
        throw "valueIsTooHighException";
    } else {
        value = input;
        response = value;
    }
} catch (e if e == "valueIsTooLowException") {
    response = "Input is too low, please enter a number above 1.";
} catch (e if e == "valueIsTooHighException";) {
    response = "Input is too high, please enter a number below 9.";
} catch (e) {
    response = "An unknown error occurred";
} finally {
    response += " Enter a new value?";
}
```

Figure 33: A multiple catches with catch-all example

Unfortunately only Mozilla based UserAgents like Mozilla FireFox, SeaMonkey, Camino web browsers, ActiveState's Komodo IDE and Komodo Edit Code Editors, etc, support multiple catches. It seems it is too difficult to implement in Apple's WebKit, Opera's Presto, KDE's KHTML and no surprise Microsoft's Trident.

Regular Expressions

You can do advanced matches of data or parts of data using regular expressions (or regexp). A regular expression literal is surrounded by forward slashes:

```
var reg1 = /zyx/;
```

Figure 34: A basic regular expression literal

This assigns a match for any instance of 'zyx' in the string or variable that you would later perform a search on.

Alternatively you can use the RegExp object:

```
var reg1 = new RegExp("zyx");
```

Figure 35: A basic regular expression object

If you know the particular regular expression will not change then it is better to use the literal version as this will be compiled and the compiled version will be used at any time the regular expression is referred to.

A collection of special characters are usually used within regular expressions:

Starts with (circumflex ^)

Indicates the search phrase to the right is the beginning of the text.

Ends with (dollar sign \$)

Browser Scripting - Legend Scrolls

Indicates the search phrase to the left is the end of the text.

Any Single Character (dot .)

Indicates any one character that is not a dot.

Or (Vertical Bar |)

Provides 'this or this' ability.

A character group (opens with a square bracket and closes with an end square bracket [])

Allows you to provide a collection of characters to be the possible character or characters such as [a-zA-Z0-9] is a single letter or digit.

A negative character group (opens with a square bracket and circumflex and closes with an end square bracket [^])

Allows you to provide a collection of characters that will not be the character or characters such as [^7b@] is a single character that is not a number 7 or a letter b or an at sign.

Zero or more times (asterisk or star *)

Indicates that the character or group to the left occurs zero or more times.

One or more times (plus sign +)

Indicates that the character or group to the left occurs one or more times.

Zero or one (question mark ?)

Indicates that the character or group to the left occurs zero times or once.

A specific number of times (number within braces {8})

Indicates that the character or group to the left occurs that many times.

A least a specific number of times (number and comma within braces {8,})

Indicates that the character or group to the left occurs at least that many times.

A range (number and comma and number within braces {8,16})

Indicates that the character or group to the left occurs between the first number of times and the second number of times such as between 8 and 16 times.

A Number (backslash and lowercase letter d \d)

Indicates a number digit, 0 to 9.

Not a number (backslash and uppercase letter d \D)

Browser Scripting - Legend Scrolls

Indicates it is anything but a digit.

A word (backslash and lowercase w `\w`)

Indicates a word.

Not a word (backslash and uppercase w `\W`)

Indicates it is anything but a word.

A whitespace (backslash and lowercase s `\s`)

Indicates a whitespace which is a space, a newline, a tab, etc.

Not a whitespace (backslash and uppercase s `\S`)

Indicates it is anything but a whitespace.

For instance, to find 'zyx' in the string "This is the zyxarmophic character with properties of the alterzyx effect." you use the `exec()` RegExp method or the `match()` String method.

```
var re = /zyx/;
var inhere = re.exec("This is the zyxarmophic character with properties of the alterzyx effect.");
```

```
var txt = new String("This is the zyxarmophic character with properties of the alterzyx effect.");
txt.match("/zyx/");
```

Figure 36: regexp

Or to simply see if 'hat' is in the string "The house sat on me hat" you can use the `test()` RegExp method or the `search()` String method.

```
var re = /hat/;
var inhere = re.test("The house sat on me hat");
```

```
var txt = new String("The house sat on me hat");
txt.search("/hat/");
```

Figure 37: regexp test

It would return `true`.

Instead of a string as the parameter of String's `split()` method you can use a regular expression.

Testing if 'Snow' is at the beginning of a string you use the circumflex character (`^`):

```
var re = /^Snow/;
var here = re.test("Snow Leopard has excellent accessibilities.");
```

Figure 38: Testing if a word is at the start

Testing if 'abilities.' is at the end of a string you use the dollar character (`$`):

```
var re = /abilities\.$/;
var here = re.test("Snow Leopard has excellent accessibilities.$");
```

Browser Scripting - Legend Scrolls

Figure 39: Testing if a word and a full stop is at the end

As a dot counts as a special character in regexp as a any single character except a full stop, we need to escape the dot using the backslash – making the dot non-special.

Testing if the string is exactly the text "Snow Leopard has excellent accessibilities." you use both the circumflex and dollar characters:

```
var re = /^Snow Leopard has excellent accessibilities\.$/;  
var here = re.test("Snow Leopard has excellent accessibilities.");
```

Figure 40: Testing if a string has the exact text

More examples:

```
// Testing the letter a, one or more c's followed by an e exists  
var re = /ac+e/;  
var here = re.test("Snow Leopard has excellent accessibilities.");
```

```
// If there is zero or more words  
var re = /\w*/;  
var here = re.test("Snow Leopard has excellent accessibilities.");
```

```
// If there is at least 2 c's or at least 2 s's  
var re = /c{2,}|s{2,}/;  
var here = re.test("Snow Leopard has excellent accessibilities.");
```

```
// If there is one or more numbers  
var re = /\d+/;  
var here = re.test("Calculating the 33rd angle of the 48th m1038i");
```

```
// If there is at least one number followed by 2 letters that are only the letters s, t, h, r, d  
and /or n  
var re = /\d+[sthrdn]{2}/;  
var here = re.test("Calculating the 33rd angle of the 48th m1038i");
```

Figure 41: More regexp examples

Wrapping parts of the regexp in parenthesis will make it remember the actual part of the string it matches which is useful when doing replaces with the String objects `replace()` method.

```
var txt = new String("This tin has changed colour");  
var result = txt.replace(/(\s[a-zA-Z])i([a-zA-Z]\s)/, "$1i$2");
```

Figure 42: String regexp replace

The above example matches a space followed by a letter, `(\s[a-zA-Z])`, which results as ' t' and is the first parenthesis in the regexp so is saved in `$1`; followed by the letter `i`; followed by the second parenthesis matching a letter and a space, `([a-zA-Z]\s)`, matching 'n ' and saved in `$2`. You can then use the backreferences (`$1`, `$2`, etc) in `replace()`'s second parameter. Essentially replacing ' tin ' as ' tan '.

Browser Scripting - Legend Scrolls

There are a few Flags in regexp that modify the search perspective.

Global (`g`)

Matches more than one instance in the string.

Case Insensitive (`i`)

Performs a case insensitive search.

Multiple Line (`m`)

Makes the circumflex not only match the very beginning of the entire string but every beginning of a line within a multiple line string (contains `\n` characters). Also makes the dollar sign not only match the very end of the entire string but every end of a line within a multiple line string.

You can also use more than one flag together.

```
var re = /ac+e/gi;  
var here = re.test("Snow Leopard has excellent acCessibilities.");
```

```
var re = new RegExp("ac+e", "gi");  
var here = re.test("Snow Leopard has excellent acCessibilities.");
```

Figure 43: Using regexp flags

Webpages

Outputting the answers of calculations, functions, conditions, variables and other features from the script into the main flow of a webpage can be done in a number of ways. The way a script interacts with the main flow of a webpage is by using the Document Object Model or DOM (more information about the DOM in the Modelling The Document Objects article).

In the DOM, the markup of the webpage is a hierarchy of webpage element objects such as the `window` object and the `document` object.

Key `window` methods include `print()` which triggers sending the contents of the webpage to a physical printer or to a software printer such as a PDF converter depending on the print dialog options the user choose at the time.

The `document` object for HTML (HyperText Markup Language) and XHTML (eXtensible HyperText Markup Language) webpages is actually an `HTMLDocument` object. Properties of this object include `title` to get the document title, `URL` providing the URI of the webpage, `body` to provide the body element object (`HTMLBodyElement`) and `documentElement` which provides the top `html` element object (`HTMLHTMLElement`).

Key `document` methods include `getElementById()` with the value of the `id=""` attribute as the parameter. It obtains the actual element object that has that `id=""` attribute value. `getElementsByName()` and `getElementsByTagName()` both get an array of element objects with the same name as the method

Browser Scripting - Legend Scrolls

parameter such as `var paras = document.getElementsByName("p");` results in an array of paragraph element objects (HTMLParagraphElement objects). You can then select a particular child element object from this array (HTMLCollection object) with the index of the child element object (first child object is zero) via the `item()` method such as:

```
var parasColl = document.getElementsByName("p");
parasColl.item(0);
```

Figure 44: Get an HTMLElement via `getElementsByName()` and `item()`

or by the body element:

```
var docbody = document.body;
docbody.item(0);
```

Figure 45: Get an HTMLElement via `body` and `item()`

Once you've obtained the particular HTMLElement object, one way to output a variable, function or whatever from the script is to use the `innerHTML` or `outerHTML` properties of the HTMLElement object. For instance assigning strings, numbers, variables to a paragraph object replaces the paragraph's contents with the string / number / variable.

```
var paraObj1 = parasColl.item(0);
paraObj1.innerHTML = "The fruit is a" + fruit[3];
```

Figure 46: Display using `innerHTML`.

For `outerHTML`, it replaces the element contents and the element itself with the assigned value.

```
paraObj1.outerHTML = "<ul><li>" + fruit[3] + "</li></ul>";
```

Figure 47: Display using `outerHTML`.

The last example replacing the entire paragraph with an HTML unordered list.

Or you can put the output in a DOM Text Node and add the Text Node to the HTMLElement such as:

```
var txtout1 = document.createTextNode("The fruit is" + fruit[3]);
paraObj1.appendChild(txtout);
```

Figure 48: Display using DOM Text Node

Also some HTMLElements have a `value` property such as adding text to the fourth `<textarea></textarea>` element:

```
var taColl = document.body.getElementsByTagName("textarea");
var textObj4 = taColl.item(3);
textObj4.value += " The height is " + product1.dimensions["height"];
```

Figure 49: Display using `value`

The `document` object also has a couple of old methods that only work in HTML documents (they would throw Document Well Formed Errors on XML Documents including native XHTML webpages): `write()` and `writeln()`.

```
document.write("<p>This is an outdated way of adding this HTML code.</p>");
```

Browser Scripting - Legend Scrolls

Figure 50: Old document.write.

The `document.writeln()` is the same as above but automatically adds a newline character on the end. Again these methods are only available for backwards compatibility.

Browser Scripting - Legend Scrolls

This article and others are available online and in other document formats at:

<http://www.legendscrolls.co.uk/webstandards/>

Author: Peter Davison from [Legend Scrolls](#).

Copyright ©2009-2010 Legend Scrolls and Peter Davison.

The Globe icon from Crystal Project Icons: LGPL, Copyright © [Everaldo](#).

All rights reserved.