# The City Lit Institute

##### **Department of Computing**

## Keeley Street, Covent Garden, London WC2B 4BA

### **COURSE : LEARN TO WRITE JavaScript**

### 

**LECTURER** : **ALEXANDER ADU- SARKODIE**

MSc. Telematics (IT & Telecom), MSc. Eng., Dip. Russ. Lang., Teach. Cert, AMIAEng (UK), MBCS (UK), MIfL(UK)

**Biopic:** [**http://www.goldhawk-college.com/istudio/**](http://www.goldhawk-college.com/istudio/)

**Email:** [**aadusarkodie1@gmail.com**](mailto:aadusarkodie1@gmail.com)

**Blog :** [**https://www.blogger.com/profile/14800490193632788559**](https://www.blogger.com/profile/14800490193632788559)

**Content**

**Synopsis**

**Introduction**

**Modules**

**Part I**

* **Understanding JavaScript**
* JavaScript and the Browser Wars
* Setting up a Web Test System
* **Performing Operations**
* The <script> tag
* Rules and Syntaxes
* The Message box & The Console Log
* Keywords
* Variables
* Data Types and Escape Sequence
* String manipulations
* Functions
* Event Handlers & Event Delegation
* Closure
* Global Pollution & Hoisting
* Unobtrusive JavaScript and the DOM
* Breakpoints
* Constructors
* **Conditional Statements & The Window Object**
* The WHILE loop
* The FOR loop
* The CONTINUE and BREAK statements
* The IF, ELSE IF and ELSE clauses
* Window properties
* **Widgets, Applications, Frameworks & Libraries**

* 1. **Unobtrusive JavaScript, and the Document Object Model**

**Document Object model** (DOM) provides:

* An **object model**, and alongside it
* an **API** (Application Programme Interface)

for a document. One of the best ways to visualize the DOM for a webpage is to use the DOM Inspector that comes with Mozilla or Chrome browsers.

Within the document object model all page elements are placed in a **tree like hierarchy**.

When you are working with the DOM, the ideal way to later **access** an element on the page is by giving it a unique **identifier**, or **ID**. For a group of elements you can use a **className** or common **attribute**s. Then, you can use the **DOM method** below to access elements and properties of a node:

*document.getElementById().*

This method then **accesses** the given element and enables you to go on from there: **Modify** the element, **append** sub-elements or otherwise **navigate** through the DOM tree.

The DOM API support methods to not only access elements in the DOM tree, but also **add** and **remove** elements. Therefore it is possible to modify virtually anything on the page.

The JavaScript you may know has already been separated from the mark-up to a certain degree. The functions that do all the work are contained in an external file. The **problem** lies with the **in-line handlers**.

Using an attribute like ***onclick*** in the mark-up is just as inefficient as using the ***style*** attribute in CSS as an in-line directive (as some of you may know). It would be much better if we could use a hook, like ***class***or ***id***, **to tether the behaviour to the mark-up without intermingling it**. This is how the mark-up could indicate that a links in a page with class of assigned value **popup** should have the popUp or overlay function executed when it is called to action (**CTA**):

<a href= “example.html” class= “popup”>example</a>

This is entirely possible. Events do not have to be handled in the mark-up. **You can attach an event to an element in an external JavaScript file**. The tricky part is figuring out **which** **element** should have the event attached.

If you want to attach an event to an element with unique *id*, you can simply use *getElementById;* i.e

***getElementById(id).event= action;***

With multiple elements, you can use a combination of ***getElementsByTagName* , *getAttribute*  and *getElementsByClassName*** to attach events to elementswith specific attributes.

Below is the procedure, for opening up a series of pop-up links from within a page with common class name **popup**.

i) Make an **array** of all the links in the document

ii) **Loop** through this array

iii) If a link has the **class "popup**", execute this **behaviour** when the link is **clicked**

iv)Pass the **value** of the links href attribute to the **popUp** function.

v) **Cancel** the **default** behaviour so that the link isn't **followed in the original window**.

And this is how it should look like;

var links= document.getElementsByTagName("a"); //make an array of link

for (var i=0; i < links.length;i++) { //loop through

if(links[i].getAttribute("class")=="popup"){ //Note use of == operator

links[i].onclick=function() {

openUp(this.getAttribute("href"));

return false; //Cancel default behaviour

}

}

}

Now the **connection** between the **links** and the **behaviour** that should occur when the links are clicked has been **moved out** of the mark-up and into the external JavaScript file. This is **unobtrusive JavaScript**.

As example again, rather than adding the onchange attribute explicitly, the relevant element(s) are simply identified, for example by class, id or some other means in the markup:

<input type="text" name="date" class=”sname” id="date" />

A script that runs when the page is first loaded into the browser can then look for the relevant element(s) and set them up accordingly:

Next fine tune the code using wrapper functions to load into browser on on-load. Note the **object model** is a property of the **window object** and so it will inherit the windows **on-load event handler**

**The unobtrusive solution is to register the necessary event handlers programmatically, rather than inline.**

Finally, don’t forget to check for **graceful degradation**. Add some code to test for DOM.

Ex 27

<html lang="en-gb">

<head>

<title>oJS 5</title>

<script language=”javascript” type="text/javascript" src="popstuff.js"></script>

</head>

<body>

<a href="poppage1.html" class ="popup">popup page1</span></a>

<a href="poppage2.html" class ="popup">popup page2</span></a>

<a href="poppage3.html" class="popup">popup page3/span></a>

</body>

</html>

**popstuff.js**

window.onload = prepareLinks; //referencing the function.

function prepareLinks() {

var links= document.getElementsByTagName("a");

for( var i=0; i<links.length;i++) {

if(links[i].getAttribute("class")=="popup"){

links[i].onclick=function() {

openUp(this.getAttribute("href"));

return false;

}

}

}

}

function openUp(winURL) {

window.open(winURL,"popup","width=320, height=480");

}

**7.1 DOM element properties**

### Element properties

**Properties Description**

**attributes[]** Returns an array containing all the attributes defined for the element in question, including custom attributes. IE6 returns not just attributes explicitly defined, but those of the element's internal DTD as well. In Firefox, attributes[] work more as expected, returning only user defined attributes, and even reflect changes done by scripting to an attribute. Each attribute[] element returned supports a **name** and **value** property to retrieve additional information about the attribute.

Ex 28

var imageattributes=document.getElementById("myimage").attributes;

imageattributes[0].name //name of the first attribute of "myimage"

imageattributes[0].value //value of the first attribute of "myimage"

imageattributes.**getNamedItem**("src").value //value of the "src" property of "myimage"

eg. <img src= “antelope.gif” alt=”The antelope” id=”ant” />

**childNodes[]** Returns an array of all of the child nodes of an element as **objects**. Use the properties "**nodeName**" and "**nodeType**" to retrieve additional information about a node.

Ex 29

//access some <ul> element  
var mylist=document.getElementById("mylist")  
for (i=0; i<mylist.childNodes.length; i++) {  
 if (mylist.childNodes[i].nodeName=="LI") {  
 //do something

}  
}

**Html**

<div>

<h2>Hey</h2>

<p>What you doing?!</p>

<ul id= “myList”>

<li>1</li>

<li>2</li>

<li>3</li>

</ul>

</div

**className** Returns the CSS class attribute of an element.

Ex 30

document.getElementById("test").className="class1" //Assign the class "class1" to element  
document.getElementById("test").className +=" class2" //Assign an additional "class2" class to element

**clientWidth** A **cross browser** property that returns the **viewable** width

of the content on the page, not including **borders**,

**margins**, or **scrollbars** (overflowing content).

Ex 31

var pagewidth=document.body.clientWidth

**clientHeight** A **cross browser** property that returns the viewable height

of the content on the page, not including borders,

margins, or scrollbars (overflowing content).

**innerHTML** A **cross browser** (non W3C DOM) property that lets you

easily change the HTML contents of an element. Generally, this property can only be invoked **after** the document has **fully loaded** (Synchronous processes). Quite useful in synchronous processes.

Ex 32

<p><b>Old paragraph text</b></p>  
<script type="text/javascript">  
 window.onload=function() {  
 document.getElementsByTagName("p")[0].innerHTML="<b>new paragraph text</b>";  
 }  
</script>

**firstChild** Returns a reference to the first child of an element (below).

**nodeName** Returns a string indicating the name of the node, in

the case of elements, its tag name. Returned value is in uppercase.

Ex 33

if (document.getElementById("test").firstChild.nodeName=="DIV") {  
 alert("This is a DIV");

}

**nodeType** Returns an integer indicating the type of a node.

**Style** References the style object of an element,

in turn accessing and modifying individual style attributes' values.

Ex 34

document.getElementById("test").style.backgroundColor="yellow";

**nodeValue** Read/write property that reflects the value of a node. For text nodes, the content of the node is returned, while for attribute nodes, the attribute value. Null is returned for Document and element nodes.

Use this property to alter the contents of a text

or attribute node.

Ex 35

<body>  
<div id="test">Old text</div>  
  
<script type="text/javascript">  
 if (document.getElementById("test").firstChild.nodeName=="#text") {  
 document.getElementById("test").firstChild.nodeValue="New text";  
 }  
</script>

**7.2 DOM – Adding Elements**

Ex 36

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

<head>

<title>DOM 3 - Removing elements</title>

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

initAddItem: function addItem() {

var list = document.getElementById("list");

var newNode = document.createElement("li");

var newTextNode = document.createTextNode("Hey, appended this text to the dynamically created list. Isn't this great! I love DOM");

newNode.appendChild(newTextNode);

list.appendChild(newNode);

list.insertBefore(newNode,list.firstChild);

}

window.onload=addItem;

</script>

</head>

<body>

<ul id="list"><li>item</li></ul>

</body>

</html>

**DOM - Changing complete HTML Fragments**

Ex 37

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

<head>

<title>innerHtml -changing complete HTML Fragments </title>

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

var nr = 1;//this is a counter

function addItem() {

var list = document .getElementById("list");// get ID of the list item

nr++;

var newNode = "<li>item" + nr + "</li>";

list.innerHTML += newNode;

}

</script>

</head>

<body onload="addItem();">

<ul id="list">

<li>item 1</li>

</ul>

</body>

</html>

**Generate Dynamic Object (template)**

Ex 38

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

<head>

<title>Styled created table from JavaScript Data - 2</title>

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

//Author : Alexander Adu-Sarkodie

//To be used with above instruction ONLY

function createTable(data) {

var table = document.createElement("table");

table.style.border = "1px solid #ffcc33";

var thead = document.createElement("thead");

thead.style.padding = "5px";

var tr = document.createElement("tr");

for (var i = 0; i < data[0].length; i++) {

var th = document.createElement("th");

th.style.border = "2px solid #ff0000";

var newText = document.createTextNode(data[0][i]);

th.appendChild(newText);

tr.appendChild(th);

}

thead.appendChild(tr);

table.appendChild(thead);

var tbody = document.createElement("tbody");

for (var i =1; i < data.length; i++){

var tr = document.createElement("tr");

for (var j = 0; j < data[i].length; j++) {

var td = document.createElement("td");

td.style.padding = "5px";

td.style.border = "2px solid #00ff00";

var newText = document.createTextNode(data[i][j]);

td.appendChild(newText);

tr.appendChild(td);

}

tbody.appendChild(tr);

}

table.appendChild(tbody);

return table;

}

window.onload = function() {

var table = createTable ([

["1","2","3","4"],

["One","Two","Three","Four"],

["Un","Deux","Trois","Quatre"],

["eins","zwet","dret","vier"]

]);

document.body.appendChild(table)

}

</script>

</head>

<body>

</body>

</html>

#### Event Delegation and Triggers

JavaScript **events** are the **bedrock** of all interactivity on web page .

In **traditional** event handling you **add** or **remove** event handlers from each element

as needed.

However, event handlers can potentially cause **memory leaks** and **performance**

**degradation** - the more you have, the greater the risk. JavaScript event delegation is a simple

technique by which you add a **single** event handler to a **parent element** in order to

avoid having to **add event handlers to multiple child elements**.

## How it works

Event delegation makes use of two features of JavaScript events: event **bubbling** and **target** element . **When an event is triggered on an element, for example a mouse click on a button, the same event is also triggered on all of that element’s ancestors**. This process is known as event **bubbling**; the event **bubbles up from the originating element to the top of the DOM tree.**

The **target** element of any event is the **originating element**, the button in our example, and is stored in a **property** of the **event object**.

**Using event delegation it’s possible to add an event handler to an element, wait for an event to bubble up from a child element and easily determine from which element the event originated.**

**In short, bubbling is the notion that for instance a clicked element registers a click, then the event "echoes" up through the nodes in the DOM to the top level -- however, you can grab that event and determine the originating source or "target" object on the page.**

Ex 39

In the example below, when the event bubbles up to the UL element, you check the event object's target property to gain a reference to the actual clicked node.  Here's a very basic JavaScript snippet which illustrates event delegation:

// Get the element, add a click listener...

document.getElementById("parent-list").addEventListener("click",function(e) {

// e.target is the clicked element!

// If it was a list item

if(e.target && e.target.nodeName == "LI") {

// List item found! Output the ID!

console.log("List item ",e.target.id.replace("post-")," was clicked!");

}

});

Ex 40

$(document).ready(function() {

$('table').click(function(event) {

var

$thisCell,

$tgt = $(event.target);

if ($tgt.is('td')) {

$thisCell = $tgt;

}

else if ($tgt.parents('td').length) {

$thisCell = $tgt.parents('td:first');

}

// now do something with $thisCell. Say execute the l });

});

Ex 41

(To be completed in class).

Now let's have a parent DIV with many children but all we care about is an A tag with the "classA" CSS class :

// Get the parent DIV, add click listener...

document.getElementById("myDiv").addEventListener("click",function(e) {

// e.target was the clicked element

if(e.target && e.target.nodeName == "A") {

// Get the CSS classes

var classes = e.target.className.split(" ");

// Search for the CSS class!

if(classes) {

// For every CSS class the element has...

for(var x = 0; x < classes.length; x++) {

// If it has the CSS class we want...

if(classes[x] == "classA") {

// Bingo!

console.log("Anchor element clicked!");

// Now do something here....

}

}

}

}

});

Event delegation is also a great way to avoid **crippling** the user's browser when you're working with a huge document. As another example , if you have a table with thousands of cells, and you want something to happen when the user clicks on one, you won't want to attach a click handler to every single one of them . Instead, you can attach the click handler to **a single table element** and use **event.target** to **pinpoint the cell that is being clicked**. Below is how a jQuery program will handle such an event.

### **Conclusions about Event Delegation**

* Is easier to assign
* Can consolidate all events into a nicer centralized package which distributes functionality from one set of events
* Persists after the DOM has loaded, and if it is modified
* Uses less memory footprint in your browser window
* Consequently, may perform better
* On a large scale desktop-like app, it will perform better

#### The Window Object, Statements & Loops

The window object represents a window containing a DOM document; the document property points to the DOM document loaded in that window.

**Statements** are used in JavaScript to **progress the execution** of the JavaScript application. They may define **loops within the code or be simple terms to evaluate**.

We will be covering conditional testing and the different types of loops.

9.1 While Loops

In this example we will make sure that the **user enters his or her name at all cost.** If the evaluation **returns true then the code in the statement block will be executed**. After the code has **been executed** the test expression will **again be evaluated and the loop will continue until the loop returns false.**

Ex 42

<html>

<head>

<title>code checking name entry</title>

</head>

<body>

<script language=”javascript” type= “text/javascript”>

var myname = " ";

myname = **prompt** (" Enter your name here, please. " **,** myname )

**while** (myname = = “” || myname = = "undefined" ) {

myname = **prompt** ( "Please try again typing your name." )

}

**document . write** ("Hello " + myname)

</script>

</body>

</html>

Note : We need to look at the use of the variable **myname.** First we give it null value and then call this *initialising* the variable :

**myname = " "**

Then we use the **variable within the prompt method to receive the user's input into the variable**

myname = **prompt** (" Enter your name here, please. " **,** myname )

Note that if the user does not enter a name, our variable **myname** will have the value *null*. If that happens we have to ask the user again to enter a name . We can do this by using the **while** loop

**while** (myname = = “” || myname = = "undefined" ) {

myname = **prompt** ( "Please try again typing your name." )

}

Note once again the **while** keyword which is followed by a condition (inside the parenthesis) which controls the loop. As long as the condition is true the loop will keep going around executing the statements that are included within { }. In this example the loop will keep asking the user for a name as long as he/she does not enter a name.

Ex 43

<html>

<head>

<title>code checking name entry</title>

</head>

<body>

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

var firstName = '';

var lastName = '';

firstName = prompt('Enter your first name',firstName);

while (firstName == ‘’ || firstName == 'undefined') {

firstName = prompt('Try again entering your first name');

}

if (firstName != ' ' && firstName != 'undefined') {

lastName = prompt('Enter your last name',lastName);//this is a more rigid test

while (lastName == ‘’ || lastName == 'undefined') {

lastName = prompt('Try again entering your last name');

}

}

//finally out put data to browser

document.write('Hello ' + firstName + ' ' + lastName);

</script>

</body>

</html>

Ex 44

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

<head>

<title>JavaScript Calendar</title>

<script type="text/javascript">

var month\_array = new Array();

month\_array[0] = "January";

month\_array[1] = "February";

month\_array[2] = "March";

month\_array[3] = "April";

month\_array[4] = "May";

month\_array[5] = "June";

month\_array[6] = "July";

month\_array[7] = "August";

month\_array[8] = "September";

month\_array[9] = "October";

month\_array[10] = "November";

month\_array[11] = "December";

document.write('<select name="day">');

var i = 1;

while ( i <= 31 ) {

document.write('<option value=' + i + '>' + i + '</option>');

i++;

}

document.write('</select>');

document.write('<select name="month">');

var i = 0;

while ( i <= 11 ) {

document.write('<option value=' + i + '>' + month\_array[i] + '</option>');

i++;

}

document.write('</select>');

document.write('<select name="year">');

var i = 1900;

while ( i <= 2005 ) {

document.write('<option value=' + i + '>' + i + '</option>');

i++;

}

document.write('</select>');

</script>

</head>

<body>

</body>

</html>

Ex 45

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

<head>

<title>Break</title>

<script type="text/javascript">

document.write("<p><b>Example of using the break statement:</b></p>");

var i = 0;

for (i=0; i<=10; i++) {

if (i==3){break;} //The counter will stop at 3

document.write("The number is " + i);

document.write("<br />");

}

</script>

</head>

<body>

</body>

</html>

Ex 46

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

<head>

<title>Continue</title>

<script type="text/javascript">

document.write("<p><b>Example of using the continue statement:</b><p>");

var i = 0;

for (i=0; i<=10; i++) {

if (i==3){continue} // The counter will skip 3

document.write("The number is " + i);

document.write("<br />")

}

</script>

</head>

<body>

</body>

</html>

* 1. Properties and methods of the window object

The properties and methods of the window.document object enable a web page document to be changed at runtime.

Using JavaScript to **assign new values to properties of the window.document object causes the** **web browser to update the appearance of the web page in line with the new values**. This can be simply illustrated by **assigning new values** to the **document.bgColor** and **document.fgColor** properties to change the **background and foreground colours of a page**.

In the example below, the setcolour( ) function runs when the **user pushes the button and the page appearance is changed**:

Ex 47

<html>

<head>

<title>window properties</title>

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

function poll(){

for (var i=0; i<document.f.rad1.length; i++){

if (document.f.rad1[i].checked){

document.bgColor=document.f.rad1[i].value;

//OR fgColor

}

}

}

</script>

</head>

<body>

<div align="center"><h2>Changing background colour</h2></div>

<div align="center">

<form name="f">

<input type="radio" name="rad1" value="white">white

<input type="radio" name="rad1" value="silver">silver

<input type="radio" name="rad1" value="skyblue">skyblue

<input type="button" value="Change colours" onclick="poll()">

</form>

</div>

</body>

</html>

Ex 48

The For loop is executed till a specified condition returns false. It takes 3 arguments and looks as follows:

for (initialization; condition; increment/decrement) {

// statements

}

When the For loop executes, the following occurs:

The initializing expression is executed. This expression usually initializes **one** or **more** loop counters, but the syntax allows an expression of any degree of complexity.

The condition expression is evaluated. If the value of condition is true, the loop statements execute. If the value of condition is false, the FOR loop terminates.

The update expression increment or decrement executes. The statements execute, and control returns to step 2.

The following example generates a multiplication table 2 through 9. **Outer loop** is responsible for generating a list of ***dividends***, and **inner loop** will be responsible for generating lists of ***dividers*** for each individual number:

Ex 49

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

<head>

<title>Loops 1</title>

<script type="text/javascript">

document.write("<h1>Multiplication table</h1>");

document.write("<table border=2 width=50%");

for (var i = 1; i <= 9; i++ ) { //this is the outer loop

document.write("<tr>");

document.write("<td>" + i + "</td>");

for ( var j = 2; j <= 9; j++ ) { // inner loop

document.write("<td>" + i \* j + "</td>");

}

document.write("</tr>");

}

document.write("</table>");

</script>

</head>

<body>

</body>

</html>

Ex 50

<!DOCTYPE html>

<html>

<body>

<script>

var i;

var mycars = new Array();

mycars[0] = "Saab";

mycars[1] = "Volvo";

mycars[2] = "BMW";

for (i=0;i<mycars.length;i++) {

document.write(mycars[i] + "<br />");

}

</script>

</body>

</html>

Ex 51

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

<head>

<title>Loops 2</title>

<script type="text/javascript">

function howMany (selectItem) {

var numberSelected=0;//create a variable to store values. Set to 0

for (var i=0; i < selectItem.options.length; i++) {//**options** is DOM property of selected items

if (selectItem.options[i].selected == true)

numberSelected++;//add to the variable. Increasing the counter.

}

return numberSelected// Send or post value to function/application

}

</script>

</head>

<body>

<p>Next example creates a function containing the For statement that counts the number of selected options in a list. The For statement declares the variable i and initializes it to zero. It checks that i is less than the number of options in the Select object, performs the succeeding if statement, and increments i by one after each pass through the loop.</p>

<form name="selectForm">

<p>Choose some book types, then click the button below:</p>

<select multiple name="bookTypes" size="8">

<option selected> Classic </option>

<option> Information Books </option>

<option> Fantasy </option>

<option> Mystery </option>

<option> Poetry </option>

<option> Humor </option>

<option> Biography </option>

<option> Fiction </option>

</select>

<input type="button" value="How many are selected?" onclick="alert ('Number of options selected: ' + howMany(document.selectForm.bookTypes))">

</form>

</body>

</html>

* 1. **Conditional *if* statement**

The “**if**” keyword is used to perform the **basic conditional JavaScript test**, to evaluate an expression for a **Boolean value**. The *statemen*t **following** the evaluation will **only be executed** when the expression **returns true**. Below is the syntax for the “if” statement.

if (test expression ) {

statement to execute when true;

}

**11.2 Using The *if else* statement with *confirm***

The JavaScript **“else”** keyword can be used with an “**if**” statement to provide **alternative code to execute in the event that the test expression returns false**. This is known as **conditional branching** and has this syntax:

if (test expression ) {

do this ;

}

else {

do this ;

}

The JavaScript **else if** keyword is used to provide response to when there are a **number** of **options or tailored responses to be considered**  for evaluation by the application or program.

Ex 52

<html>

<head>

<title> using the else if </title>

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

**function** validate ( ){

**if**  ( document.myform**.**firstname**.**value = = " " ){

**alert** ( "You must enter your first name" ) **; //** checks for name entry

return **;**

}

**else if** ( document.myform.surname.value = = “ ” ){ //checks for surname entry

alert (“You must enter your surname.”) ;

return;

}

submitnnow = **confirm** ("Do you want to submit now ?"); //disclaimer or flag

**if** (submitnow){

document**.**myform**.**submit ( ) **;**

}

**else** {

return **;**

}

//document.getElementById(“send”).onclick = validate;

}

</script>

</head>

<body>

<form name ="myform">

<h1>booking form</h1>

first name : <input name = "firstname">

<br />

surname :<input name ="surname">

<input type = "button" id= “send” value = "book now" **onclick** = "validate ( )">

</form>

</body>

</html

The function we have written validates the name and surname. This forces the user to enter his name and surname in the form.

The function code is included in the HEAD section. This is always the case when we are building our own functions.

We call the function by using its name **validate().** This function is invoked or triggered by the event handler **onclick.** Used in the context here as an *attribute***.** In our case the validate function will be used to check the name, and surname.

**11.3 Email Validation**

Ex 53

This simple email validation script tests for the presence of the **‘@**’ character in an email address

<html>

<head>

<title>Email validation</title>

<script type="text/javascript">

function validation(){

if(document.f.email.value.indexOf("@") == -1) {

alert("Email address is absent or incorrect");

return false;

}

}

</script>

</head>

<body>

<div align="center">

<form name="f" method="post" action="mailto:abc@aol.com" enctype="text/plain" **onsubmit**="**return** validation()">

Enter your email address:<input type="text" name="email" value="">

<input type="submit" value="submit">

</form>

</div>

</body>

</htm>

**A more robust validation script**

Ex 54

<html>

<head>

<title>email validation</title>

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

<!--

function send\_if\_valid() {

if(document.f.email.value == "") {

fail("you must enter your email address");

}

else if(document.f.email.value.indexOf("@") == -1) {

fail("No '@' in the address");

}

else {

var addr = document.f.email.value.split("@");

if (addr[0].length < 1 ) {

fail("User address absent");

}

else if (addr[1].indexOf(".") == -1 ) {

fail("No dot");

}

else if (addr[1].length < 3) {

fail("Domain incorrect");

}

else {

document.f.submit();

}

}

}

function fail(msg) {

alert("Email Address Error : \n" + msg);

}

//-->

</script>

</head>

<body>

<form name="f" method="post" action="#">

<!-- <form name="f" method="post" action="mailto:bob@bob.com"> -->

<p>Please enter email address:

<input name="email" type="text" value="">&nbsp;<input type="button" value="submit form" onclick="send\_if\_valid()">

</p>

</form>

</body>

</html>

# Making AJAX Calls

One core feature of 21st century web applications is their fantastic responsiveness.

Behind this amazing feat of human cleverness is the coming together of a few technologies collectively known as **AJAX**. You need acess to a server to write and retrieve AJAX applications.

In this lesson we will learn:

* what **AJAX stands for** and **what it is all about**;
* how to **make an AJAX request**;
* how to **handle an AJAX response**.

## What's AJAX all about?

The acronym AJAX stands for **Asynchronous JavaScript And XML**.

It is a combination of internet standards made up of:

* **standards-based presentation** using **HTML and CSS**;
* **dynamic display** using the **DOM**;
* **data interchange** and manipulation using **XML or JSON**;
* **asynchronous** data retrieval using the **XMLHttpRequest object**;
* **JavaScript** magic to orchestrate the whole process.

In non-AJAX web applications, the interaction between servers and clients can be a tedious business:

1. a user action from the client sends a request to the web server via HyperText Transfer Protocol (HTTP);
2. the web server receives the request and processes it by invoking server-side scripts, database data, etc., and sends a response back to the client via HTTP;
3. the client browser receives the response and loads the entire updated page.

Having to go from browser to server and back again each time a user requests some piece of data from the server, in addition to undergoing an entire page refresh at each update, can be quite stressful on servers and on users alike.

AJAX helps in at least 2 respects:

* Avoiding Page Refresh
* Operations are performed asynchronously, means that during the time that it takes for the server to respond, the page is not locked and the user can still interact with the website.

## How do I make an AJAX request?

An **AJAX request** is made using the **XMLHttpRequest object** and its **open() and send() methods**. This is supported by all major browsers. However, older browsers, namely older varsions of Microsoft Internet Explorer (vesions 5 and 6), support an **ActiveXObject**. This little hurdle is easily overcome by testing for feature support in the script.

The **open(retrievalMethod, url, bool)** method has 3 arguments:

1. **retrievalMethod**: this can either be a **GET** (used to fetch data from the server), or a **POST** (used to send data to the server);
2. **url**: this is the location where the data is made available. It can be a text file, an XML document, or a server-side script that processes data coming from a database;
3. **bool**: this is a true/false value. If it's false the request is made synchronously, if it's true the request is made asynchronously, which is what we usually want.

The XMLHttpRequest object has an **onreadystatechange** property that deals with the response from the server. This proceeds over the following 5 stages:

* 0) the **request is uninitialized** because **open() has not been called**;
* 1) the **request is specified**, but **send() has not been called yet**;
* 2) the **request is being sent**, because now **send() has been called**;
* 3) the **response is being received, but not yet completed**;
* 4) the **response is complete and data is available for manipulation and display**.

Upon completion (stage 4), the XMLHttpRequest object's **status property** gets assigned an **HTTP status code** that describes the result of the request as follows:

* **200**: success!
* **401**: unauthorized - authentication is required and was not provided;
* **403**: forbidden - the server refuses to respond;
* **404**: not found - the requested resource cannot be found.

In our next 3 examples we will see how you can use AJAX to retrieve three different types of data:

* Text : http://html.net/tutorials/javascript/lesson18\_ex1.html
* XML: <http://html.net/tutorials/javascript/lesson18_ex2.html>
* AJAX calls with JQuery: http://html.net/tutorials/javascript/lesson21.php
* JSON: <file:///C:/Users/alexs/Documents/DigiTek/AjaxResponses/render-ext-data/loading-images-2.html>

**JavaScript Object Prototypes**

Every JavaScript object has a prototype. The prototype is also an object.

All JavaScript objects inherit their properties and methods from their prototype.

Objects created using an object literal, or with new Object(), inherit from a prototype called Object.prototype.

## Creating a Prototype

Example:

<html>

<body>

<p id="demo"></p>

<script>

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

var myFather = new person("John", "Doe", 50, "blue");

var myMother = new person("Sally", "Rally", 48, "green");

document.getElementById("demo").innerHTML =

"My father is " + myFather.age + ". My mother is " + myMother.age;

</script>

Sometimes you want to add

* new properties (or methods) to an existing object.
* add new properties (or methods) to all existing objects of a given type.
* add new properties (or methods) to an object prototype.

## Adding a Property to an Object

Adding a new property to an existing object is easy:

<html>

<body>

<p id="demo"></p>

<script>

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

var myFather = new person("John", "Doe", 50, "blue");

var myMother = new person("Sally", "Rally", 48, "green");

myFather.nationality = "English";

document.getElementById("demo").innerHTML =

"My father is " + myFather.nationality;

</script>

</body>

</html>

## Adding a Method to an Object

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

var

firstName,

lastName,

age,

eyeColor;

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

var myFather = new person("John", "Doe", 50, "blue"); // create an object

var myMother = new person("Sally", "Rally", 48, "green");

myFather.name = function() {

return this.firstName + " " + this.lastName;

};

document.getElementById("demo").innerHTML =

"My father is " + myFather.name();

</script>

</body>

</html>

## Adding Properties to a Prototype

You cannot add a new property to a prototype the same way as you add a new property to an existing object, because the prototype is not an existing object.

To add a new property to a constructor, you must add it to the **constructor function**:

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

this.nationality = "English";//note

}

var myFather = new person("John", "Doe", 50, "blue");

var myMother = new person("Sally", "Rally", 48, "green");

document.getElementById("demo").innerHTML =

"My father is " + myFather.nationality + ". My mother is " + myMother.nationality;

</script>

</body>

</html>

## Adding Methods to a Prototype

Your constructor function can also define methods:

<html>

<body>

<p id="demo"></p>

<script>

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

this.name = function() {

return this.firstName + " " + this.lastName

};

}

var myFather = new person("John", "Doe", 50, "blue");

document.getElementById("demo").innerHTML =

"My father is " + myFather.name();

</script>

</body>

</html>

## Using the **prototype** Property

The JavaScript prototype property allows you to add new properties to an existing prototype:

<html>

<body>

<p id="demo"></p>

<script>

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

person.prototype.nationality = "English";

var myFather = new person("John", "Doe", 50, "blue");

document.getElementById("demo").innerHTML =

"My father is " + myFather.nationality;

</script>

</body>

</html>

The JavaScript prototype property also allows you to add new methods to an existing prototype:

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

person.prototype.name = function() {

return this.firstName + " " + this.lastName

};

var myFather = new person("John", "Doe", 50, "blue");

document.getElementById("demo").innerHTML =

"My father is " + myFather.name();

</script>

</body>

</html>

**The call() and apply() methods**

One very common thing when writing Javascript is knowing when to use call and when to use apply.

Let's look at some ways we might want to use them:

**var** person1 = {name: 'Marvin', age: 42, size: '2xM'};

**var** person2 = {name: 'Zaphod', age: 42000000000, size: '1xS'};

**var** sayHello = **function**(){

alert('Hello, ' + **this**.name);

};

**var** sayGoodbye = **function**(){

alert('Goodbye, ' + **this**.name);

};

Writing the following code:

sayHello();

sayGoodbye();

will give errors , or just unexpected results . This is because both functions rely on their scope for the this.name data, and calling them without explicit scope will just run them in the scope of the current window.

So how do we scope them?

sayHello.call(person1);

sayGoodbye.call(person2);

sayHello.apply(person1);

sayGoodbye.apply(person2);

All four of these lines do exactly the same thing. The run sayHello or sayGoodbye in the scope of either person1 or person2.

Both call and apply perform very similar functions: they execute a function in the context, or scope, of the first argument that you pass to them. Also, they're both functions that can only be called on other functions. You're not going to able to runperson1.call(), nor does it make any sense to do so.

The difference is when you want to seed this call with a set of arguments. Say you want to make a say() method that's a little more dynamic:

**var** say = **function**(greeting){

alert(greeting + ', ' + **this**.name);

};

say.call(person1, 'Hello');

say.call(person2, 'Goodbye');

So that's call for you. It runs the function in the context of the first argument, and subsequent arguments are passed in to the function to work with. So how does it work with more than one argument?

**var** update = **function**(name, age, size){

**this**.name = name;

**this**.age = age;

**this**.size = size;

};

update.call(person1, 'Slarty', 200, '1xM');

No big deal. They're simply passed to the function if it takes more than one parameter.

The limitations of call quickly become apparent when you want to write code that doesn't (or shouldn't) know the number of arguments that the functions need… like a dispatcher.

**var** dispatch = **function**(person, method, args){

method.apply(person, args);

};

dispatch(person1, say, ['Hello']);

dispatch(person2, update, ['Slarty', 200, '1xM']);

So that's where apply comes in - the second argument needs to be an array, which is unpacked into arguments that are passed to the called function.

So that's the difference between call and apply. Both can be called on functions, which they run in the context of the first argument. In call the subsequent arguments are passed in to the function as they are, while apply expects the second argument to be an array that it unpacks as arguments for the called function.

# Array.prototype.filter()

## Summary

The **filter()** method creates a new array with all elements that pass the test implemented by the provided function.

## Syntax

arr.filter(callback[, thisArg])

### Parameters

callback

Function to test each element of the array. Return true to keep the element, false otherwise.

thisArg

Optional. Value to use as this when executing callback.

## Description

filter calls a provided callback function once for each element in an array, and constructs a new array of all the values for which callback returns a true value. callback is invoked only for indexes of the array which have assigned values; it is not invoked for indexes which have been deleted or which have never been assigned values. Array elements which do not pass the callback test are simply skipped, and are not included in the new array.

callback is invoked with three arguments:

1. the value of the element
2. the index of the element
3. the Array object being traversed

If a thisArg parameter is provided to filter, it will be passed to callback when invoked, for use as its this value.  Otherwise, the value undefined will be passed for use as its this value.  The this value ultimately observable by callback is determined according to [the usual rules for determining the this seen by a function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this).

filter does not mutate the array on which it is called.

The range of elements processed by filter is set before the first invocation of callback. Elements which are appended to the array after the call to filter begins will not be visited by callback. If existing elements of the array are changed, or deleted, their value as passed to callback will be the value at the time filter visits them; elements that are deleted are not visited.

## Examples

### Example: Filtering out all small values

The following example uses filter to create a filtered array that has all elements with values less than 10 removed.

function isBigEnough(element) {

return element >= 10;

}

var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);

// filtered is [12, 130, 44]