# 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/profile>

**LinkedIn:** <https://www.linkedin.com/profile/preview?locale=en_US&trk=prof-0-sb-preview-primary-button>

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

**Blog :** <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
* Using Breakpoints
* Constructors
* **Conditional Statements & The Window Object**
* The IF, ELSE IF and ELSE statements
* The WHILE loop
* The FOR loop
* The CONTINUE and BREAK statements
* The SWITCH statement
* Chaining statements in an application
* -Window properties
* The CALL statement
* The APPLY statement
* Visualisation (d3js and Highchart)
* **Best Practices**
* **Frameworks & Libraries**
* **Introduction to Object Oriented JavaScript and ECMA2015**

**Learn To Write JavaScript**

**JavaScript for Web development**

In order to provide **meaningful information** to corporate intranet users and to **compete for business** on the World Wide Web, organisations must build websites that provide **services**, are **effective and responsive**. Java Script is a **programming language** that provides a **powerful tool** for **rapidly developing highly** **interactive, dynamic web sites**.

This course provides in-depth hands on **experience writing client-side JavaScript**. You will learn how to **integrate** JavaScript **into your web pages** and create an **interactive** and **dynamic web site**.

**You will learn to:**

**Development Skills**

* Write and debug client – side JavaScript that works across multiple browsers
* Create dynamic and interactive web pages
* Add interactivity forms that capture and validate user input
* Personalise pages
* Use Event Handlers (triggers), Event Delegation and Calls To Action (CTA) on JavaScript functions
* Write Unobtrusive JavaScript, manipulate DOM and AJAX operations
* Build and work with JavaScript Libraries
* Use Best Practices

**What will I achieve?**

**Mastery Skills**

* Write and debug JavaScript functions
* Create interactive, self validating forms
* Implement interactive user interface using modal and dialogue boxes.
* Event Delegation techniques.
* Manipulate DOM and AJAX operations

# Synopsis

JavaScript is a **Client side interpreted programming language**. A language, that **runs** within the **web browser (**client**).**

Its interpreteris **embedded** inside web browser software, such as **Microsoft Internet Explorer** and **Mozilla** browser family, **Chrome**, **Safari** and **Opera** to mention a few.

The language **grants access** to the current page and lets the script determine **properties** of the client, **redirect** the user to another page, **access** cookies and do much more.

The main principle behind **embedded** scripting within an HTML document is to **control the web page from the client-side rather than from the server** in contrast to CGI scripting. In this way the web documents can be read by the browser’s “**JavaScript engine**” whenever the document is loaded into the browsers window.

The JavaScript language contains many impressive features but for **security** reasons it cannot **read or write** (**copy**) files, with the exception of “**cookie**” files that store a small amount of data (maximum size: 4 kilobytes). Referred to as **Cross Domain Policy.**

Combining JavaScript with knowledge of DOM enables powerful creation of dynamic html effects.

1. **Understanding JavaScript**

**1a. JavaScript and the Browser Wars**

The birth of JavaScript was in **September 1995**, originally by Brendan Eich with Netscape version 2 named **Mocha** and later re-named **LiveScript** after marketing deal with Java. Microsoft began to promote and integrate JavaScript and later into their Internet Explorer (IE) browsers. For legal reasons called it **JScript**. The worse was to follow as Microsoft begun computing specific **incompatible library features** and **characteristics** for their JScript engine without **legal commercial licensing.**

In **1997** the European Computer Manufacturers Association( **ECMA** 262) the standard was published, and JavaScript was the **first** implementation. The standard only **specified** the **language** and **not features**. This escalated more inconsistencies and the browser war continued. Netscape **scrapped** the idea of releasing browser **version 5** and decided to start it all over again with **Netscape 6** (the Mozilla project). All this helped Internet Explorer to expand its market share reaching over **90%** by **early 2000**. The Mozilla project gave birth to **Firefox** and other flavours of the **Mozilla** browser derivatives (Epiphany, Camino, Galen etc).

Until IE7, Internet Explorer was far away from supporting the latest JavaScript 1.5 released in 2005. With IE7+ and FF 2.0 + released, developers are now much more comfortable developing for **cross browser compatibility**.

Other browsers, by the way support JavaScript as well. Differences are subtle, but still can be extremely annoying when one is developing a **browser – agnostic** web application. Among the browsers that currently support JavaScript are the following:

* Internet Explorer
* Mozilla and all derivatives (Firefox, Epiphany, Camino and Galeaon)
* Opera
* Konqueror
* Safari
* Chrome

The current modern war is both on **platforms** and **technology**. Some companies are:

* Abandoning platforms or slowly deploying their technology to cloud cluster platforms.
* The modern ecology of the web-space continues to be challenging, and there is evangelisation of more broad support for the basic structures of the web, using cloud based architectures, but for the bleeding-edge this is an organic process.

Note, not all browsers may have JavaScript enabled. Though, with the avalanche of smart devices, and increased bandwidth associated with them JavaScript is now a de-facto and by default activated on these devices. Test your website and applications built with JavaScript first **enabled**, and then **disabled**. **Core** **functionality should be maintained when JavaScript is** **disabled** (you may have to provide alternate solution or strategy for core functionality without JavaScript), and the rendering of your pages should **degrade gracefully**.

Until recent advancement of platform computing environments like NodeJS that allows JavaScript to be deployed from the back end which has changed the eco-system of the technologies implementation as an isomorphic tool, generally speaking, JavaScript should be used for **progressive enhancement**. Of late **CTM** (**cut the mustard)**  tests (courtesy of the British Broadcasting Corporation - BBC) is used to provide enhanced user experience for modern mobile phones that have higher CPUs and bandwidth.

**The Future**

* JavaScript **performance** will continue to be an important part of each browser's competitive advantage.
* Google has a big interest in JavaScript speed and performance on different browsers due to their **cloud computing** strategies.
* Microsoft is catching up to their competitors' on how well JavaScript performs on IE. Providing development and deployment cloud platforms and services to the public.
* Popular new web standards such as HTML5 and CSS3 need JavaScript and its associated Frameworks to join them together.

.

#### 2. Performing Operations and Making Statements

### **2.1 The *<script>* tag**

The **<script>** tag enables us to **include** scripts within an HTML document. It contains the **script block** and usually shares the **HEAD** section of the document with the **TITLE** and the **META** tags. The **LANGUAGE** attribute defines the **type** of scripting language we are using. If omitted the **SCRIPT** tag will assume that **JavaScript** is used. Also, the type **attribute** must be placed in the script tag and must be set to “**text/javascript**”.

When a browser loads a document it **reads,** or **parses,** the code **sequentially**. Top, to bottom. So placing thescript blockin the **head** of the document other than in the **body** ensures that the JavaScript code is **parsed before the rest of the HTML content.** However, it should be noted that depending on the **timely requirement and intervention of resources needed by an application on the page, including DOM elements**, the script can equally be put at the **bottom** of the embedded html document.

A script block looks like this:

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

**<!- -**

**//- - >**

**</script>**

**nb :** The actual code will be placed in between the line beginning with <! -- and //- ->. **These lines are there to hide the script from old browsers.**  This syntax is hardly needed these days with the volley of new devices in the market and enhanced browser capabilities.

Alternatively, the script can be written into an **external** JavaScript file and referenced back into the html document using the **src** attribute within the script tag. This is simply a text file saved with a “**.js**” file extension and containing **no** html tags**.** Wherethey **occur,** they should be **escaped** appropriately in order not tostop the page from rendering properly or throwing errors.

In the script tag which is placed in the header will now have a **src** attribute pointing the browser to the **url** (Universal Resource Locator) of the JavaScript file from its repository. As below, assuming this file is called librarycode.js. The closing **</script>** tag **must** persist.

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

**src = “librarycode.js”></script>**

**Note: The space in between the opening and closed tags MUST be empty.**

### **Syntax rules. [ 5]**

* There is a **semi-colon** at the end of the JavaScript statement to comply with JavaScript syntax rules. This makes it a self contained statement.
* JavaScript is a **case sensitive** language and that “**ALERT**”, “**Alert**” and “**alert**” are all different words.
* All JavaScript **keywords** are ***identifier names*** and are specially, **reserved words**. They are in **lowercase** only.
* **Spaces, tabs and new lines** are collectively known as **white space**  and are completely **ignored** by JavaScript **parser**, so the code maybe **formatted** and **indented** and its appearance made more **human-readable**.
* It is often useful to add **comments** or **annotation** to JavaScript code as **explanation**, or when an **update** needs to happen. The parser sees any text between // and the **end** of that line as a **single-line** comment, which it ignores. Also any text, on **one** or **more lines** between **/\*** **\*/** is ignored.

**2.1 The Message Box.**

The **alert** command - Generating the *message box;*

Ex. 1

<html>

<head>

<title>scripting 1</title>

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

**alert** ( "Hello ! this is an alert message, and my first JavaScript programme. ");

</script>

</head>

<body>

</body>

</html>

The message box is achieved by calling the ***alert*** command in the code. When the document is loaded in the browser; the “**JavaScript engine**” implements the instructions contained in the code. **The code causes the browser to open a *grey box* (an alert box) bearing the message contained within the quotation marks inside the brackets**. The quotation marks do not appear and are used as **delimiters** for the string of characters representing the message. The quotation marks **MUST** always be there.

Using the console tab and break points in modern browsers, you can debug your code.

**Note**: Using escapes: Alternate single, and double quotes: Both are interchangeable and must be consistent.

Ex. 2

<html>

<head>

<title>Alternating with single and double quotes</title>

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

**alert** ( "Hello ! this is an alert message for the ‘cool’ guys ");

**// console.log** ( "Hello ! this is an alert message for the \”cool\” guys ");

</script>

</head>

<body>

</body>

</html>

## 2.2 Keywords

These **should not be used when using identifier names** and are part of the JavaScript language syntax (reserved words). In a professional editor all keywords will come out **bolded.**

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

**2.3 Variables**

A **variable** is a **place** where **data** is **stored** and can be **manipulated** within a JavaScript **program**. Any **letter, digit** or the **underscore** (i.e. a[A], 1[n], \_ ) may be used to name a variable. The only exception is **not to begin with a digit**.

In the following example a new variable called ***message*** has been created using the JavaScript keyword “**var**”. The text string is stored inside the variable named *message*. This variable is used in the **call** to the JavaScript alert function that opens an **alert dialogue box** displaying the **string** that is stored in the variable. Note the call also known as ***initialisation***to the variable name inside alert has no **delimiters.**

Ex 4

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

<!- -

var message = “First JavaScript variable”;

alert (**message**);

//- - >

</script>

**2.4** **Data types & Escape Sequences**

Variables in JavaScript can be used to store **numbers, text strings** or **boolean** values **(true/false)**.This is unlike certain programming languages like C++ and Java where specific data types have to be declared and can store data of declared type.

The example below creates variables with initial values of the **three different** data types supported by JavaScript. These three values are *first* passed to the *alert ( )* function, then the JavaScript keyword **typeof**  is used to **return** their data types for display in the alert dialog box.

The syntax “**+\n+**” inside the brackets displays the output for each value on a **new line** in the alert dialog box.

Ex 5

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

var

a = 0.06,

b = “Is JavaScript easy?”,

c = false;

alert (**typeof** a + “\n” + **typeof** b + “\n” + **typeof** c );

</script>

**Escape sequence**:

When a **character** in a string is **preceded** by the **backslash** character **“\”** there is a **special effect** on the **character immediately following it**. This is referred to as **escape sequence** as it allows the *letter* **following it to escape recognition but present a meaning as part of the JavaScript syntax**. The table below lists more escape sequences:

|  |  |  |
| --- | --- | --- |
| \b | Backspace |  |
| \f | Form feed |  |
| \n | New line |  |
| \r | Carriage return |  |
| \t | Tab |  |
| \' | Single quote that will not terminate a string |  |
| \" | Double quote that will not terminate a string |  |
| \\ | Single backslash character |  |

The escape sequence “ **\”** “ (be careful) is useful to incorporate quotation marks within a string without the string terminating early as shown in the example below.

alert (“We all say **\”** Javascript is handy **\”** “);

**3.1 Arithmetical Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| Arithmetical Operators |  |  |  |
|  |  |  |  |
| Operator | Operation |  |  |
| + | Addition(numeric), and concatenates strings–depending on operands |  |  |
| - | Subtraction |  |  |
| \* | Multiplication |  |  |
| / | Division |  |  |
| % | Modulo – returns remainder |  |  |
| ++ | Increment (alters the given value by 1 and returns resulting value) |  |  |
| - - | Decreases a given value by 1 and returns result. |  |  |

**3.2 Comparison Operators**

|  |  |
| --- | --- |
| Operator | Comparative test |
| = = | Equality |
| ! = | Inequality |
| > | Greater than |
| < | Less than |
| === | Matches equality & Type |
| > = | Greater than or equal to |
| < = | Less than or equal to |

The equality operator “= =” **compares two operands and will return true if both are equal in value**. It could be ***strings containing the same characters in same position or numbers***. **Boolean operands that are both, true, or both false, are equal.**

Conversely the “!=” operator returns true if two operands are not equal using the same rules as the “ = =”

“Greater than” operators compare two operands and will return true if the first operand is greater in value than the second.

“Less than” operators compares two operands, and will return true if the first operand is less in value than the second.

Adding the “ =” operator after a “greater than” or “less than” operator makes it also return true if the two operands are exactly equal in value.

The “greater than” operator “>” is frequently used to test the value of a countdown value in a loop.

4. The String object

The String object is used to manipulate a stored piece of text.

1. Returning the **length** of a string
2. **Length** is a property of the string object

Ex. 6

<html>

<head>

<title> Returning the length of a string</title>

</head>

<body>

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

var txt = "Hello World!";

document.write(txt.**length**);

</script>

</body>

</html>

*ii)* ***Styling*** *strings*

Ex. 7

<html>

<head>

<title> *Styling strings* </title>

</head>

<body>

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

var txt = "Hello World!";

document.write("<p>Big: " + txt.**big()** + "</p>");

document.write("<p>Small: " + txt.**small()** + "</p>");

document.write("<p>Bold: " + txt.**bold()** + "</p>");

document.write("<p>Italic: " + txt.**italics()** + "</p>");

document.write("<p>Fixed: " + txt.**fixed()** + "</p>");

document.write("<p>Strike: " + txt.**strike()** + "</p>");

document.write("<p>Fontcolor: " + txt.**fontcolor("green**") + "</p>");

document.write("<p>Fontsize: " + txt.**fontsize(6)** + "</p>");

document.write("<p>Subscript: " + txt.**sub()** + "</p>");

document.write("<p>Superscript: " + txt.**sup()** + "</p>");

document.write("<p>Link: " + txt.**link("http://www.w3schools.com")** + "</p>");

document.write("<p>Blink: " + txt.**blink()** + " (does not work in IE, Chrome, or Safari)</p>");

</script>

</body>

</html>

*iii ) How to convert a string to lowercase or uppercase letters.*

The **toLowerCase()** and **toUpperCase** methods

Ex. 8

<html>

<head>

<title> *How to convert a string to lowercase or uppercase letters* </title>

</head>

<body>

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

var txt="Hello World!";

document.write(txt.toLowerCase() + "<br />");

document.write(txt.toUpperCase());

</script>

</body>

</html>

iv) *How to search for a specified value within a string.*

The **match()** method

Ex. 9

<html>

<head>

<title> *How to search for a specified value within a string****.***</title>

</head>

<body>

<script type="text/javascript">

var str="Hello world!";

document.write(str.match("world") + "<br />");

document.write(str.match("World") + "<br />");

document.write(str.match("worlld") + "<br />");

document.write(str.match("world!"));

</script>

</body>

</html>

v) *How to replace a specified value with another value in a string.*

**Replace characters in a string – replace()** method

Ex. 10

<html>

<head>

<title> *How to replace a specified value with another value in a string.*</title>

</head>

<body>

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

var str="Visit Microsoft!";

document.write(str.replace("Microsoft","W3Schools"));

</script>

</body>

</html>

vi) *How to return the position of the first found occurrence of a specified value (a character or a word) in a string.*

**The indexOf()** method

NB: If no value is found in the evaluation, -1 is returned. The count position begins from zero.

Ex. 11

<html>

<head>

**<**title**>**

*How to return the* ***position****, of the first found occurrence of a specified value in a string*

</title>

</head>

<body>

<p>Note this can be the **position** of a **character** or position of a **word**</p>

<script type="text/javascript">

var str="Hello world!";

console.log(str.indexOf("d") + "<br />");

console.log (str.indexOf("WORLD") + "<br />");

console.log (str.indexOf("world"));

</script>

</body>

</html>

vii) *charAt (position)*

charAt() simply returns the character at the specified position. The count position begins from zero. For example:

var message="internet"

//alerts "n"

console.log(message.charAt(1)) // Will return character at second position if found.

viii) *slice (start, end)*

As the name implies, slice() extracts out a substring from the string as determined by the starting and ending points of its parameters.

Note the position for the last argument is not returned.

var text="excellent"

text.slice(0,4) //returns "exce"

text.slice(2,4) //returns "ce"

ix) *split(delimiter)*

The split() method.

split() cuts up a string into pieces, using the delimiter as the point to cut off, and stores the results into an array. Consider the following message:

var message="Welcome to JavaScript Kit";

var word=message.split(" ");

If the expression above is parsed:

//word[0] will contain "Welcome",

word[1] contains "to" etc

Variable word instantly becomes an array that holds the individual words. This is so because we used a space (" ") as the delimiter, which also is what's separating each word.

Following on from this, here's the same message again, manipulated by split() using a different delimiter this time:

var message="Welcome to JavaScript Kit";

var word=message.split("l");

//word[0] contains "We",

word[1] contains "come toJavaScript Kit"

The split() method is often used to parse values stored inside a cookie, since they are by default separated by semicolons (;), a set delimiter.

x) substring (from, to)

This method simply returns the substring beginning with the "from" parameter (included as part of the substring), and ending with "to" (NOT included as part of substring). It behaves just like the slice() method seen earlier. For example:

var text="excellent";

text.substring(0,4) //returns "exce"

text.substring(2,4) //returns "ce"

xi) Date() Object

The Date object is used to work with dates and times. Date objects are created with the Date() constructor.

All dates are calculated in milliseconds from 01 January, 1970 00:00:

There are four ways of initiating a date:

new Date() // current date and time  
new Date(milliseconds) //milliseconds since 1970/01/01  
new Date(dateString)  
new Date(year, month, day, hours, minutes, seconds, milliseconds)

Most parameters above are optional. Not specifying, causes 0 to be passed in.

Once a Date object is created, a number of methods allow you to operate on it. Most methods allow you to get and set the year, month, day, hour, minute, second, and milliseconds of the object, using either local time or UTC (universal, or GMT) time.

**Set Date**

We can easily manipulate the date by using the methods available for the Date object.

In the example below we set a Date object to a specific date (14th January 2010):

Ex. 12

var myDate=new Date();  
myDate.setFullYear(2010,0,14);

And in the following example we set a Date object to be 5 days into the future:

Ex 13

var myDate=new Date();  
myDate.setDate(myDate.getDate()+5);

Ex3

**Compare two dates**

The Date object is also used to compare two dates. The following example compares today's date with the 14th January 2100:

Ex 14

var x=new Date();  
x.setFullYear(2100,0,14);  
var today = new Date();  
  
if (x>today){  
 alert("Today is before 14th January 2100. Sales hasn’t started yet");

//  
}  
else{  
alert("Today is after 14th January 2100. Sales begins at 10am");  
}

**5. The Array Object**

The Array object is used to store **multiple values** in a **single variable**. The object has **predefined properties** and **methods:**

**The Array Object Properties**

|  |  |
| --- | --- |
| **Property** | **Description** |
| Constructor | Returns the function that created the Array object's prototype |
| Length | Sets or returns the number of elements in an array |
| Prototype | Allows you to add properties and methods to an Array object |

**The Array Object Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| concat() | Joins two or more arrays, and returns a copy of the joined arrays |
| indexOf() | Search the array for an element and returns it's position |
| join() | Joins all elements of an array into a string |
| lastIndexOf() | Search the array for an element, starting at the end, and returns it's position |
| pop() | Removes the last element of an array, and returns that element |
| push() | Adds new elements to the end of an array, and returns the new length |
| reverse() | Reverses the order of the elements in an array |
| shift() | Removes the first element of an array, and returns that element |
| slice() | Selects a part of an array, and returns the new array |
| sort() | Sorts the elements of an array |
| splice() | Adds/Removes elements from an array |
| toString() | Converts an array to a string, and returns the result |
| unshift() | Adds new elements to the beginning of an array, and returns the new length |
| valueOf() | Returns the primitive value of an array |

**Creating an Array**

An array can be created in three ways.

**Regular Array**

var myCars=new Array();   
myCars[0]="Saab";   
myCars[1]="Volvo";  
myCars[2]="BMW";

**Condensed Array**

var myCars=new Array("Saab","Volvo","BMW");

**Literal Array**

var myCars=["Saab","Volvo","BMW"];

**Access an Array.**

You refer to an element in an array by referring to the **index** number.

This statement access the value of the first element in myCars. You assign the returned value to a new variable.

var name=myCars[0];

And the following statement modifies the first element in myCars:

myCars[0]="Opel";

You can have different objects in an array. You can have functions in an Array. You can have arrays in an Array (Multi-dimensional Array):

myArray[0]=Date.now;  
myArray[1]=myFunction();  
myArray[2]=myCars;

**Array methods and properties**

Examples;

var x=myCars.length // the number of elements in myCars;  
var y=myCars.indexOf("Volvo") // the index position of "Volvo";

6. Functions

A function is a named sub-routine.It is a **piece of JavaScript code that can be** **executed once or many times** by the JavaScript application. **It is a way of packaging your JavaScript commands so you can easily reuse them every time you need the same piece of functionality implemented in your application**.

A function is created by using the JavaScript  *keyword* **“function”** followed by a given **identifier name**. The name must be **unique** ***through out the whole function***. The name is followed by a *pair* of **open plain brackets**, then a pair **of curly brackets** to contain the code to be executed. The function can be **called anywhere within the document** to execute **the statement** that it contains.

In this lesson, you will learn:

* what **functions** are and how they are used in JavaScript;
* how to **retrieve information from functions**;
* how to **give information to functions**;
* what **variable scope** is all about.

**This suggests that there are 2 distinct phases to a function:**

* The phase where thefunction is declared (created)**;**
* The phase where thefunction is called (used)**.**

There are a number of ways of declaring function types. Two most common ones are;

var artist = function () { //**anonymous function**

do something;

}

and

function getName() { // named function

do something;

}

Ex 15

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

<!--

function call\_alert( ) {

alert( “my first javascript function”);

}

//-->

</script>

The **onload** attribute **(this is an event handler).** We canuseevent handlers to trigger the function when the document **is loaded**.

*Note the script is placed* ***in the header*** *in this case.*

<body **onload= “call\_alert( )”>**

<body onload = “call\_alert( )”>

Or unobtrusively as;

**window.onload = call\_alert;**

|  |
| --- |
|  |
| Note the absence of the parenthesis in this case.  Ex 16    <html>  <head>  <meta charset="utf-8" />  <title> Package your JavaScript Code with Functions</title>  </head>  <body>  <h1> Package your JavaScript Code with Functions</h1>    <div>    <h2>Today's date is:</h2>    <span id="calendar"></span>    <input type="button" id="myButton" value="Get Date" onclick="showDate();" />    </div>    <script type="text/javascript">  function showDate() {    var today = new Date();  var myCalendar = document.getElementById("calendar");  var myButton = document.getElementById("myButton");  myCalendar.innerHTML = today.toDateString();  myButton.value = "Well done!";  }  </script>  </body>  </html> |

Ex 17: Word counter:

<form name="wordcount">

<textarea rows="12" name="wordcount2" cols="38" wrap="virtual"></textarea>

<br>

<input type="button" value="Calculate Words" onClick="countit()"> <input type="text" name="wordcount3" size="20">

</form>

<script type="text/javascript">

function countit(){

var formcontent = document.wordcount.wordcount2.value;

var content = formcontent.split(" ");

document.wordcount.wordcount3.value = content.length;

}

</script>

**6.2 Function Argument (Abstraction)**

The **plain brackets** that follow the name of all functions **may** be used to **contain data for use in the code to be executed**. This data is called the function “**argument**”. In the example that follows below the function call passes a string to the argument named “str” in the call\_alert( ) function for use in the code to be executed.

***You can use or reference this same block of code in different pages by assigning different string values. The function becomes universal, and can easily be propagated to different page environments.* *Re-using code (DRY – Don’t Repeat Yourself) and enhancing re-usability. This allows maintenance from a single source.***

function call\_alert(str ) {

alert( str);

}

}

<body onload = “call\_alert (‘passed value’)”>

It is important to note that string contained **inside the call** is enclosed in **single** quotes to differentiate it from the double quotes used to contain the **entire call**. Improper use will cause error. Assuming double quotes were used all over then there will be ***early termination*** after the second double quote and an error message will be produced.

Ex 18

<html>

<head>

<title>toString</title>

<script>

function myFunction(){

var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.toString();

var x=document.getElementById("demo");

x.innerHTML=fruits;

}

window.onload = myFunction; //function call.

</script>

</head>

<body>

<p>The toString() method converts an array into a String and returns the result.

<br />

Note: The returned string will separate the elements in the array with commas.

</p>

<p id="demo">Click the button to convert the array into a String.</p>

</body>

</html>

Ex 19

<html>

<head>

<title>Push</title>

<script>

var fruits = ["Banana", "Orange", "Apple", "Mango"];

function myFunction(){

fruits.push("Kiwi")

var x=document.getElementById("demo");

x.innerHTML=fruits;

}

window.onload = myFunction;

</script>

</head>

<body>

<p>The push() method adds new items to the end of an array, and returns the new length.

</p>

<p id="demo">Click the button to add a new element to the array.</p>

<!-- <button onclick="myFunction()">Try it</button> -->

</body>

</html>

Ex 20

<html>

<head>

<title>Pop</title>

<script>

var fruits = ["Banana", "Orange", "Apple", "Mango"];

function myFunction(){

fruits.pop();

var x=document.getElementById("demo");

x.innerHTML=fruits;

}

</script>

</head>

<body>

<p>Removes the last element of an array, and returns the array</p>

<p id="demo">Click the button to remove the last array element.</p>

<button onclick="myFunction()">Try it</button>

</body>

</html>

**6.2 Writing External JavaScript Files**

Below is a simple external JavaScript document created by moving the script from the example above into a new separate document. This document is saved with a **.js** extension and should not have any html tags. Where they occur, they must be escaped appropriately so the script does not break the web document. Use known debugging methods to rectify any problems that persists. This document should neither have the **script** **tag** nor any **hiding syntaxes**. **Commenting** is allowed.

Using external js files allows for easier maintenance from a single source. Code can be reused and easily portable. Saving time to re-write the code into any page that may require similar routines (DRY – **D**on’t **R**epeat **Y**ourself).

1. *mylibrary.js:*

//begin

function call\_me (str){

alert(str);

}

//end

1. *Secondly embed a script tag in the calling page and point to the javascript file using the src attribute in the tag*. *As seen in the two examples below.*

Ex 21

<html>

<head>

<title>Using External JavaScript Files 1</title>

<script language="javascript" type="text/javascript" **src=" mylibrary.js">**</script>

</head>

<body onload="call\_me('This text is personalised to this page. It is using external js file to load')">

</body>

</html>

Ex 22

<html>

<head>

<title>Using External JavaScript Files</title>

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

</head>

<body onload="call\_me('This is a new second page. It is using the same external js file to load as the first. An example of code re-usability')">

</body>

</html>

**6.3 Nested functions and closures**

Variables declared **inside a function** are called **local variables** and can only be used **within that function**. Conversely, “**global**” variables are declared **outside functions and can be accessed by any function within the same document.**

You can nest a function within a function. The **nested** (inner) function is **private** to its containing (outer) function. It also forms a ***closure***.

**A closure is an expression (**typically a function**) that can have free variables together with an environment that binds those variables (that "closes" the expression).**

Since a nested function is a closure, this means that a nested function can "**inherit**" the arguments and variables of its containing function. In other words**, the inner function contains the scope of the outer function.**

To summarize:

* The inner function can be accessed only from statements in the outer function.
* The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

Ex 23

<html>

<head>

<title>multipleFunction:Building Calculator Function</title>

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

function addSquares(a,b) {

function square(x) {

return x \* x;

}

return square(a) + square(b);

}

//a = addSquares(2,3); // returns 13

//b = addSquares(3,4); // returns 25

//c = addSquares(4,5); // returns 41

document.write(addSquares(2,3));

` </script>

</head>

<body onload="addSquares(2,3)">

</body>

</html>

Ex 24

<script>

function sayHello2(name) { //function  
   var text = 'Hello ' + name; // statement  
   var sayAlert = function() { //publish

alert(text);

}  
  return sayAlert;

}

</script>

Ex 25

<script>

function buildList(list) {  
   var result = [];  
   for (var i = 0; i < list.length; i++) {  
     var item = 'item' + list[i];  
     result.push( function() {

alert(item + ' ' + list[i])

} );

}  
   return result;

</script>

6.4 Making Multiple Arguments

JavaScript functions may contain **multiple arguments if they are separated by a *comma***. Note the **number of arguments** defined when the function **is declared** must be **exactly matched** by the **number of arguments contained in any call to that function***. Any unused arguments in assigning data must be replaced by* ***empty string*** *or space. This is achieved by using a pair of quotes* acting as white space**.** In the example below three arguments have been declared in the function, so **any call** to it **must contain three argument values:**

***Note:***

**1. You can use this same block of code in different pages of your site by passing different string values to the onload attribute of the body. Mind very carefully that, unused arguments in the call to the function must be replaced by an empty string denoted by** *‘’* **or there will be error messages showing up. The called argument in the function call MUST be separated by commas and each value held in a delimiter.**

Ex 26

var a, b, c ; //global variables

function call\_alert ( str1, str2, str3 ) {

//these three arguments are assigned to the global variables.

a = str1;

b = str2;

c = str3;

alert ( a + b + c); //print the values (variables) stored in the argument

}

<body onload = “call\_alert (‘Great’, ‘ ’,’Javascript‘)”>

The caller passes **three strings** to the function with the value of the **second merely a space**. The function assigns the argument values to the three global variables, then calls alert ( ) to display the global values as **a single concatenated string.**

* 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 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

**Design Patterns**

* Constructor Pattern
* Module Pattern
* Revealing Module Pattern
* Singleton Pattern
* Observer Pattern
* Mediator Pattern
* Prototype Pattern
* Command Pattern
* Facade Pattern
* Factory Pattern
* Mixin Pattern
* Decorator Pattern
* Flyweight Pattern
* JavaScript MV\* Patterns
* MVC Pattern
* MVP Pattern
* MVVM Pattern
* Modern Modular JavaScript Design Patterns
* AMD
* CommonJS
* ES Harmony
* Design Patterns In jQuery
* Composite Pattern
* Adapter Pattern
* Facade Pattern
* Observer Pattern
* Iterator Pattern
* Lazy Initialization Pattern
* Proxy Pattern
* Builder Pattern

**Anti-Patterns**

If we consider that a pattern represents a best practice, an anti-pattern represents a lesson that has been learned. The term anti-patterns was coined in 1995 by Andrew Koenig in the November C++ Report that year, inspired by the GoF's book Design Patterns. In Koenig’s report, there are two notions of anti-patterns that are presented. Anti-Patterns:

Describe a bad solution to a particular problem which resulted in a bad situation occurring

Describe how to get out of said situation and how to go from there to a good solution

On this topic, Alexander writes about the difficulties in achieving a good balance between good design structure and good context:

“These notes are about the process of design; the process of inventing physical things which display a new physical order, organization, form, in response to function.…every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem”.

While it’s quite important to be aware of design patterns, it can be equally important to understand anti-patterns. Let us qualify the reason behind this. When creating an application, a project’s life-cycle begins with construction however once you’ve got the initial release done, it needs to be maintained. The quality of a final solution will either be good or bad, depending on the level of skill and time the team have invested in it. Here good and bad are considered in context - a ‘perfect’ design may qualify as an anti-pattern if applied in the wrong context.

The bigger challenges happen after an application has hit production and is ready to go into maintenance mode. A developer working on such a system who hasn’t worked on the application before may introduce a bad design into the project by accident. If said bad practices are created as anti-patterns, they allow developers a means to recognize these in advance so that they can avoid common mistakes that can occur - this is parallel to the way in which design patterns provide us with a way to recognize common techniques that are useful.

To summarize, an anti-pattern is a bad design that is worthy of documenting. Examples of anti-patterns in JavaScript are the following:

Polluting the global namespace by defining a large number of variables in the global context

Passing strings rather than functions to either setTimeout or setInterval as this triggers the use of eval() internally.

Modifying the Object class prototype (this is a particularly bad anti-pattern)

Using JavaScript in an inline form as this is inflexible

The use of document.write where native DOM alternatives such as document.createElement are more appropriate. document.write has been grossly misused over the years and has quite a few disadvantages including that if it's executed after the page has been loaded it can actually overwrite the page we're on, whilst document.createElement does not. We can see here for a live example of this in action. It also doesn't work with XHTML which is another reason opting for more DOM-friendly methods such as document.createElement is favorable.

Knowledge of anti-patterns is critical for success. Once we are able to recognize such anti-patterns, we're able to refactor our code to negate them so that the overall quality of our solutions improves instantly.

**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 run person1.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]

**IIFE** (Immediate Invoked Function Expression)

(function(){

…

});

The first pair of parentheses (function(){...}) turns the code within (in this case, a function) into an expression, and the second pair of parentheses (function(){...})() calls the function that results from that evaluated expression.

This pattern is often used when trying to avoid polluting the global namespace, because all the variables used inside the IIFE (like in any other *normal* function) are not visible outside its scope.

(function(){

// all your code here

var foo = function() {};

window.onload = foo;

// ...

})();

// foo is unreachable here (it’s undefined)

It declares an anonymous function, then calls it:

This says execute immediately.

var val = (function(){

var a = 0; // in the scope of this function

return function(x){

a += x;

return a;

};

})();

alert(val(10)); //10

alert(val(11)); //21

**List Operations**

* Map
* Filter
* Reduce

Reference - <https://github.com/kukuu/jsListOperations>

*==> A Map From List to List*

Often, we find ourselves needing to take an array and modify every element in it in exactly the same way. Typical examples of this are squaring every element in an array of numbers, retrieving the name from a list of users, or running a regex against an array of strings.

map is a method built to do exactly that. It's defined on Array.prototype, so you can call it on any array, and it accepts a callback as its first argument.

When you call map on an array, it executes that callback on every element within it, returning a new array with all of the values that the callback returned.

Under the hood, map passes three arguments to your callback:

* The current item in the array
* The array index of the current item
* The entire array you called map on

Why use *map* instead of *forEach*

There are a few important differences between the two approaches:

* Using map, you don't have to manage the state of the for loop yourself.
* You can operate on the element directly, rather than having to index into the array.
* You don't have to create a new array and push into it. map returns the finished product all in one go, so we can simply assign the return value to a new variable.

You do have to remember to include a *return* statement in your callback. If you don't, you'll get a new array filled with undefined.

Turns out, all of the functions we'll look at today share these characteristics.

The fact that we don't have to manually manage the state of the loop makes our code simpler and more maintainable. The fact that we can operate directly on the element instead of having to index into the array makes things more readable.

Using a forEach loop solves both of these problems for us. But map still has at least two distinct advantages:

* forEach returns undefined, so it doesn't chain with other array methods.
* map returns an array, so you can chain it with other array methods.
* map returns an array with the finished product, rather than requiring us to mutate an array inside the loop.

Keeping the number of places where you modify state to an absolute minimum is an important tenet of functional programming. It makes for safer and more intelligible code.

Now is also a good time to point out that if you're in Node, testing these examples in the Firefox browser console, or using Babel or Traceur, you can write this more concisely with ES6 arrow functions:

var task\_names = tasks.map((task) => task.name );

Arrow functions let us leave out the return keyword in one-liners.

*It doesn't get much more readable than that.*

Gotchas

The callback you pass to map must have an explicit return statement, or map will spit out an array full of undefined. It's not hard to remember to include a return value, but it's not hard to forget.

If you do forget, map won't complain. Instead, it'll quietly hand back an array full of nothing. Silent errors like that can be surprisingly hard to debug.

*==> Filter Out the Noise*

The next of our array operations is filter. It does exactly what it sounds like: It takes an array, and filters out unwanted elements.

Like map, filter is defined on Array.prototype. It's available on any array, and you pass it a callback as its first argument. filter executes that callback on each element of the array, and spits out a new array containing only the elements for which the callback returned true.

Also like map, filter passes your callback three arguments:

* The current item
* The current index
* The array you called filter on

filter in Practice

Let's revisit our task example. Instead of pulling out the names of each task, let's say I want to get a list of just the tasks that took me two hours or more to get done.

Just like map, filter lets us:

avoid mutating an array inside a forEach or for loop

assign its result directly to a new variable, rather than push into an array we defined elsewhere

Gotchas

The callback you pass to map has to include a return statement if you want it to function properly. With filter, you also have to include a return statement, and you must make sure it returns a boolean value.

If you forget your return statement, your callback will return undefined, which filter will unhelpfully coerce to false. Instead of throwing an error, it will silently return an empty array!

Always make sure your callbacks include an explicit return statement. And always make sure your callbacks in filter return true or false. Your sanity will thank you.

*==> Reducing Arrays*

map creates a new array by transforming every element in an array, individually. filter creates a new array by removing elements that don't belong. reduce, on the other hand, takes all of the elements in an array, and reduces them into a single value.

Just like map and filter, reduce is defined on Array.prototype and so available on any array, and you pass a callback as its first argument. But it also takes an optional second argument: the value to start combining all your array elements into.

reduce passes your callback four arguments:

* The current value
* The previous value
* The current index
* The array you called reduce on

Notice that the callback gets a previous value on each iteration. On the first iteration, there is no previous value. This is why you have the option to pass reduce an initial value: It acts as the "previous value" for the first iteration, when there otherwise wouldn't be one.

Finally, bear in mind that reduce returns a single value, not an array containing a single item. This is more important than it might seem, and I'll come back to it in the examples.

**Inheritance and Composition**

Composition over inheritance (or composite reuse principle) in object-oriented programming is the principle that *classes should achieve polymorphic behaviour and code reuse by their composition* (by containing instances of other classes that implement the desired functionality) *rather than inheritance from a base class.*

To favour composition over inheritance is a design principle that gives the design higher flexibility. It is more natural to build business-domain classes out of various components than trying to find commonality between them and creating a family tree.

*Drawbacks*

One common drawback of using composition instead of inheritance is that methods being provided by individual components may have to be implemented in the derived type, even if they are only forwarding methods. In contrast, inheritance does not require all of the base class's methods to be re-implemented within the derived class. Rather, the derived class only need to implement (override) the methods having different behaviour than the base class methods. This can require significantly less programming effort if the base class contains many methods providing default behaviour and only a few of them need to be overridden within the derived class.

*inheritance*:

class Fruit {

//...

}

class Apple extends Fruit {

//...

}

By composition, I simply mean using **instance variables that are references to other objects.** For example:

class Fruit {

//...

}

class Apple {

private Fruit fruit = new Fruit();

//...

}

*==> Name spacing*

Namespaces can be considered a logical grouping of units of code under a unique identifier. The identifier can be referenced in many namespaces and each identifier can itself contain a hierarchy of its own nested (or sub) namespaces.

In application development, we employ namespaces for a number of important reasons. In JavaScript, they help us avoid collisions with other objects or variables in the global namespace. They're also extremely useful for helping organize blocks of functionality in a code-base so that it can be more easily referenced and used.

Namespacing any serious script or application is critical as it's important to safeguard our code from breaking in the event of another script on the page using the same variable or method names we are. With the number of third-party tags regularly injected into pages these days, this can be a common problem we all need to tackle at some point in our careers.

Namespaces can be found in almost any serious JavaScript application. Unless we're working with a simple code-snippet, it's imperative that we do our best to ensure that we're implementing namespacing correctly as it's not just simple to pick-up, it'll also avoid third party code clobbering our own. The patterns we'll be examining in this section are:

* Single global variables
* Prefix namespacing
* Object literal notation
* Nested namespacing
* Immediately-invoked Function Expressions

Namespace injection

1. *Single global variables*

One popular pattern for namespacing in JavaScript is opting for a single global variable as our primary object of reference. A skeleton implementation of this where we return an object with functions and properties can be found below:

var myApplication = (function () {

function(){

//...

},

return{

//...

}

})();

Although this works for certain situations, the biggest challenge with the single global variable pattern is ensuring that no one else has used the same global variable name as we have in the page.

2. *Prefix namespacing*

One solution to the above problem, as mentioned by Peter Michaux, is to use prefix namespacing. It's a simple concept at heart, but the idea is we select a unique prefix namespace we wish to use (in this example, myApplication\_) and then define any methods, variables or other objects after the prefix as follows:

var myApplication\_propertyA = {};

var myApplication\_propertyB = {};

function myApplication\_myMethod(){

//...

}

This is effective from the perspective of decreasing the chances of a particular variable existing in the global scope, but remember that a uniquely named object can have the same effect.

This aside, the biggest issue with the pattern is that it can result in a large number of global objects once our application starts to grow. There is also quite a heavy reliance on our prefix not being used by any other developers in the global namespace, so be careful if opting to use this.

For more on Peter's views about the single global variable pattern, read his excellent post on them http://michaux.ca/articles/javascript-namespacing.

3. *Object literal notation*

Object literal notation (which we also cover in the module pattern section of the book) can be thought of as an object containing a collection of key:value pairs with a colon separating each pair of keys and values where keys can also represent new namespaces.

var myApplication = {

// As we've seen, we can easily define functionality for

// this object literal..

getInfo:function(){

//...

},

// but we can also populate it to support

// further object namespaces containing anything

// anything we wish:

models: {},

views: {

pages: {}

},

collections: {}

};

One can also opt for adding properties directly to the namespace:

myApplication.foo = function(){

return "bar";

}

myApplication.utils = {

toString:function(){

//...

},

export: function(){

//...

}

}

Object literals have the advantage of not polluting the global namespace but assist in organizing code and parameters logically. They are truly beneficial if we wish to create easily-readable structures that can be expanded to support deep nesting. Unlike simple global variables, object literals often also take into account tests for the existence of a variable by the same name so the chances of collision occurring are significantly reduced.

*In the next sample, we demonstrate a number of different ways in which we can check to see if a variable (object or plugin namespace) already exists, defining it if it doesn't.*

// This doesn't check for existence of "myApplication" in

// the global namespace. Bad practice as we can easily

// clobber an existing variable/namespace with the same name

var myApplication = {};

// The following options \*do\* check for variable/namespace existence.

// If already defined, we use that instance, otherwise we assign a new

// object literal to myApplication.

//

// Option 1: var myApplication = myApplication || {};

// Option 2: if( !MyApplication ){ MyApplication = {} };

// Option 3: window.myApplication || ( window.myApplication = {} );

// Option 4: var myApplication = $.fn.myApplication = function() {};

// Option 5: var myApplication = myApplication === undefined ? {} : myApplication;

We'll often see developers opting for Option 1 or Option 2 - they are both straight-forward to understand and are equivalent in terms of their end-result.

Option 3 assumes that we're working in the global namespace, but it could also be written as:

myApplication || (myApplication = {});

This variation assumes that myApplication has already been initialized and so it's only really useful for a parameter/argument scenario as in the following example:

function foo() {

myApplication || ( myApplication = {} );

}

// myApplication hasn't been initialized,

// so foo() throws a ReferenceError

foo();

// However accepting myApplication as an

// argument

function foo( myApplication ) {

myApplication || ( myApplication = {} );

}

foo();

// Even if myApplication === undefined, there is no error

// and myApplication gets set to {} correctly

Options 4 can be useful for writing jQuery plugins where:

// If we were to define a new plugin..

var myPlugin = $.fn.myPlugin = function() { ... };

// Then later rather than having to type:

$.fn.myPlugin.defaults = {};

// We can do:

myPlugin.defaults = {};

This results in better compression (minification) and can save on scope lookups.

Option 5 is a little similar to Option 4, but is a long-form which evaluates whether myApplication is undefined inline such that it's defined as an object if not, otherwise set to an existing value for myApplication if so.

var namespace = (function () {

// defined within the local scope

var privateMethod1 = function () { /\* ... \*/ },

privateMethod2 = function () { /\* ... \*/ }

privateProperty1 = "foobar";

return {

// the object literal returned here can have as many

// nested depths as we wish, however as mentioned,

// this way of doing things works best for smaller,

// limited-scope applications in my personal opinion

publicMethod1: privateMethod1,

// nested namespace with public properties

properties:{

publicProperty1: privateProperty1

},

// another tested namespace

utils:{

publicMethod2: privateMethod2

}

...

}

})();

The benefit of object literals is that they offer us a very elegant key/value syntax to work with; one where we're able to easily encapsulate any distinct logic or functionality for our application in a way that clearly separates it from others and provides a solid foundation for extending our code.

A possible downside however is that object literals have the potential to grow into long syntactic constructs. Opting to take advantage of the nested namespace pattern (which also uses the same pattern as its base)

This pattern has a number of other useful applications too. In addition to namespacing, it's often of benefit to decouple the default configuration for our application into a single area that can be easily modified without the need to search through our entire codebase just to alter them - object literals work great for this purpose. Here's an example of a hypothetical object literal for configuration:

var myConfig = {

language: "english",

defaults: {

enableGeolocation: true,

enableSharing: false,

maxPhotos: 20

},

theme: {

skin: "a",

toolbars: {

index: "ui-navigation-toolbar",

pages: "ui-custom-toolbar"

}

}

}

Note that JSON is a subset of object literal notation and there are really only minor syntactical differences between it and the above (e.g JSON keys must be strings). If for any reason one wishes to use JSON for storing configuration data instead (e.g. for simpler storage when sending to the back-end), feel free to.

4. *Nested namespacing*

An extension of the object literal pattern is nested namespacing. It's another common pattern used that offers a lower risk of collision due to the fact that even if a namespace already exists, it's unlikely the same nested children do.

var myApp = myApp || {};

// perform a similar existence check when defining nested

// children

myApp.routers = myApp.routers || {};

myApp.model = myApp.model || {};

myApp.model.special = myApp.model.special || {};

// nested namespaces can be as complex as required:

// myApp.utilities.charting.html5.plotGraph(/\*..\*/);

// myApp.modules.financePlanner.getSummary();

// myApp.services.social.facebook.realtimeStream.getLatest();

Note: The above differs from how YUI3 approaches namespacing as modules there use a sandboxed API host object with far less and far shallower namespacing.

We can also opt to declare new nested namespaces/properties as indexed properties as follows:

myApp["routers"] = myApp["routers"] || {};

myApp["models"] = myApp["models"] || {};

myApp["controllers"] = myApp["controllers"] || {};

Both options are readable, organized and offer a relatively safe way of namespacing our application in a similar fashion to what we may be used to in other languages. The only real caveat however is that it requires our browser's JavaScript engine first locating the myApp object and then digging down until it gets to the function we actually wish to use.

This can mean an increased amount of work to perform lookups, however developers such as Juriy Zaytsev have previously tested and found the performance differences between single object namespacing vs the "nested" approach to be quite negligible.

5. Immediately-invoked Function Expressions (IIFE)s

Earlier in the book, we briefly covered the concept of an IIFE (immediately-invoked function expression) which is effectively an unnamed function, immediately invoked after it's been defined. If it sounds familiar it's because you may have previous come across it referred to as a self-executing (or self-invoked) anonymous function, however. In JavaScript, because both variables and functions explicitly defined within such a context may only be accessed inside of it, function invocation provides an easy means to achieving privacy.

IIFEs are a popular approach to encapsulating application logic to protect it from the global namespace but also have their use in the world of namespacing.

Examples of IIFEs can be found below:

// an (anonymous) immediately-invoked function expression

(function () { /\*...\*/ })();

// a named immediately-invoked function expression

(function foobar () { /\*..\*/ })();

Examples of self-executing functions, which are quite different than IIFEs, can be found below:

// named self-executing function

function foobar () { foobar(); }

// anonymous self-executing function

var foobar = function () { arguments.callee(); }

Back to the IIFEs, a slightly more expanded version of the first IIFE example might look like:

var namespace = namespace || {};

// here a namespace object is passed as a function

// parameter, where we assign public methods and

// properties to it

(function( o ){

o.foo = "foo";

o.bar = function(){

return "bar";

};

})( namespace );

console.log( namespace );

Whilst readable, this example could be significantly expanded on to address common development concerns such as defined levels of privacy (public/private functions and variables) as well as convenient namespace extension. Let's go through some more code:

// namespace (our namespace name) and undefined are passed here

// to ensure 1. namespace can be modified locally and isn't

// overwritten outside of our function context

// 2. the value of undefined is guaranteed as being truly

// undefined. This is to avoid issues with undefined being

// mutable pre-ES5.

;(function ( namespace, undefined ) {

// private properties

var foo = "foo",

bar = "bar";

// public methods and properties

namespace.foobar = "foobar";

namespace.say = function ( msg ) {

speak( msg );

};

namespace.sayHello = function () {

namespace.say( "hello world" );

};

// private method

function speak(msg) {

console.log( "You said: " + msg );

};

// check to evaluate whether "namespace" exists in the

// global namespace - if not, assign window.namespace an

// object literal

})( window.namespace = window.namespace || {} );

// we can then test our properties and methods as follows

// public

// Outputs: foobar

console.log( namespace.foobar );

// Outputs: You said: hello world

namespace.sayHello();

// assigning new properties

namespace.foobar2 = "foobar";

// Outputs: foobar

console.log( namespace.foobar2 );

Extensibility is of course key to any scalable namespacing pattern and IIFEs can be used to achieve this quite easily. In the below example, our "namespace" is once again passed as an argument to our anonymous function and is then extended (or decorated) with further functionality:

// let's extend the namespace with new functionality

(function( namespace, undefined ){

// public method

namespace.sayGoodbye = function () {

namespace.say( "goodbye" );

}

})( window.namespace = window.namespace || {});

// Outputs: goodbye

namespace.sayGoodbye();

If you would like to find out more about this pattern, I recommend reading Ben's IIFE post for more information.

6. *Namespace injection*

Namespace injection is another variation on the IIFE where we "inject" the methods and properties for a specific namespace from within a function wrapper using this as a namespace proxy. The benefit this pattern offers is easy application of functional behaviour to multiple objects or namespaces and can come in useful when applying a set of base methods to be built on later (e.g. getters and setters).

The disadvantages of this pattern are that there may be easier or more optimal approaches to achieving this goal (e.g. deep object extension / merging) which I cover earlier in the section.

Below we can see an example of this pattern in action, where we use it to populate the behaviour for two namespaces: one initially defined (utils) and another which we dynamically create as a part of the functionality assignment for utils (a new namespace called tools).

var myApp = myApp || {};

myApp.utils = {};

(function () {

var val = 5;

this.getValue = function () {

return val;

};

this.setValue = function( newVal ) {

val = newVal;

}

// also introduce a new sub-namespace

this.tools = {};

}).apply( myApp.utils );

// inject new behaviour into the tools namespace

// which we defined via the utilities module

(function () {

this.diagnose = function(){

return "diagnosis";

}

}).apply( myApp.utils.tools );

// note, this same approach to extension could be applied

// to a regular IIFE, by just passing in the context as

// an argument and modifying the context rather than just

// "this"

// Usage:

// Outputs our populated namespace

console.log( myApp );

// Outputs: 5

console.log( myApp.utils.getValue() );

// Sets the value of `val` and returns it

myApp.utils.setValue( 25 );

console.log( myApp.utils.getValue() );

// Testing another level down

console.log( myApp.utils.tools.diagnose() );

Angus Croll has also previously suggested the idea of using the call API to provide a natural separation between contexts and arguments. This pattern can feel a lot more like a module creator, but as modules still offer an encapsulation solution, we'll briefly cover it for the sake of thoroughness:

// define a namespace we can use later

var ns = ns || {},

ns2 = ns2 || {};

// the module/namespace creator

var creator = function( val ){

var val = val || 0;

this.next = function () {

return val++

};

this.reset = function () {

val = 0;

}

}

creator.call( ns );

// ns.next, ns.reset now exist

creator.call( ns2, 5000 );

// ns2 contains the same methods

// but has an overridden value for val

// of 5000

As mentioned, this type of pattern is useful for assigning a similar base set of functionality to multiple modules or namespaces. I would however only really suggest using it where explicitly declaring functionality within an object/closure for direct access doesn't make sense.