# 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**

**Introduction**

**Synopsis**

**Modules**

* **Understanding JavaScript**
* JavaScript and the Browser Wars
* Setting up a Web Test System
* JavaScript Design Patterns
* **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
* Constructors - The date() object
* **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
* JavaScript List Operations
* **AJAX operations**
* **Websockets**
* **Best Practices**
* **Introduction to Object Oriented JavaScript**

- Classes, Objects,

- Inheritance,

- Polymorphism

- Interfaces

- Encapsulation

- Prototypes, Composition and Inheritance

- S.O.L.I.D

* **ECMA2015 - es6**
* **Frameworks & Libraries**

**-** NPM

- Embedded JavaScript Engine (ejs)

**-** REACT

**=>** Run-time messaging

**=>** Charts

=> Tabbed Navigation & Routing

- Angular2

* **Continuous Development & Continuous Integration**
* **Unit Testing versus Test Driven Development (TDD)**

**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    <!DOCTYPE html>  <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

<!DOCTYPE html>

<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

<!DOCTYPE html>

<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

<!DOCTYPE html>

<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

<!DOCTYPE html>

<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

<!DOCTYPE html>

<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

<!DOCTYPE html>

<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

<!DOCTYPE html>

<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>

<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>

<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>

<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>