JavaScript

for Developers

55244AC / MAX003AC

**Audience:**

Developers with a foundation in C# or Java who need a solid introduction to JavaScript.

**Prerequisites:**

Basic programming skills using C# or Java.

AUTHOR

Michael T Smith  
Microsoft Certified Trainer  
SharePoint / Office Servers and Services MVP  
Senior Instructor, MAX Technical Training

COPYRIGHT

© 2017 MAX Technical Training, Inc. All Rights Reserved. This manual and any training materials supplied with it are copyrighted with all rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system or translated into any language in any form by any means without the w

Portions copyright Microsmith, Inc. Used with permission.

TRADEMARKS

Brand names, company names and product names used herein are trademarks or registered trademarks of their respective companies.

MAX Technical Training, Inc.   
4900 Parkway Drive, Suite 160   
Mason, OH 45040  
513.322.8888

55244AC / MA-003AC 7/30/2017

Table of Contents

[Module 1: Basic Scripting 4](#_Toc490232424)

[JavaScript Basics 4](#_Toc490232425)

[JavaScript or ECMAScript? 4](#_Toc490232426)

[Which Version of JavaScript am I Using? 4](#_Toc490232427)

[Shims, Shivs, and Polyfills 5](#_Toc490232428)

[JavaScript 5](#_Toc490232429)

[Writing and Testing JavaScript 6](#_Toc490232430)

[Editors 6](#_Toc490232431)

[A Demonstration of Using the Browser’s F12 Tools 7](#_Toc490232432)

[Key Parts of a Script 8](#_Toc490232433)

[Useless Trivia! 21](#_Toc490232434)

[Module 2: The Document Object Model 22](#_Toc490232435)

[The DOM 22](#_Toc490232436)

[Accessing Objects from the DOM 22](#_Toc490232437)

[Document Properties 23](#_Toc490232438)

[Document Methods to Find Elements 23](#_Toc490232439)

[Responding to Events 25](#_Toc490232440)

[Common Events 25](#_Toc490232441)

[Adding Event Handler Code 25](#_Toc490232442)

[Event Bubbling / Event Propagation 26](#_Toc490232443)

[Stopping Event Propagation 27](#_Toc490232444)

[Adding Elements to the DOM 28](#_Toc490232445)

[Create HTML from Strings 28](#_Toc490232446)

[Creating New Elements 29](#_Toc490232447)

[Adding Attributes to New Elements 30](#_Toc490232448)

[Adding Text Inside of a New Element 31](#_Toc490232449)

[Removing Elements from the DOM 31](#_Toc490232450)

[Delete in Reverse Order 33](#_Toc490232451)

[Module 3: Introduction to jQuery 35](#_Toc490232452)

[jQuery 35](#_Toc490232453)

[First the negatives: 35](#_Toc490232454)

[And then the positives: 35](#_Toc490232455)

[JavaScript vs. jQuery 36](#_Toc490232456)

[Take a Peek… 36](#_Toc490232457)

[Downloading the jQuery Library 37](#_Toc490232458)

[jQuery Versions 37](#_Toc490232459)

[Linking to the Library 37](#_Toc490232460)

[“jQuery.” vs. “$.” 38](#_Toc490232461)

[Controlling When Your Code Runs 38](#_Toc490232462)

[Selecting Elements using jQuery 39](#_Toc490232463)

[Working with the Data Returned by jQuery 39](#_Toc490232464)

[Setting CSS Properties 40](#_Toc490232465)

[Running a Function Against the jQuery Return Set 40](#_Toc490232466)

[.each() 41](#_Toc490232467)

[jQuery DOM Manipulation Methods 41](#_Toc490232468)

[jQuery Effects and Animations 41](#_Toc490232469)

[jQuery Utility Methods 42](#_Toc490232470)

[jQuery Plug-ins 42](#_Toc490232471)

[Useless Trivia! 42](#_Toc490232472)

[Module 4: Ajax and Web Services 43](#_Toc490232473)

[Web Services 43](#_Toc490232474)

[HTTP Verbs 44](#_Toc490232475)

[Data Formats 44](#_Toc490232476)

[XML 44](#_Toc490232477)

[JSON 45](#_Toc490232478)

[Ajax 47](#_Toc490232479)

[Calling Web Services using JavaScript 48](#_Toc490232480)

[Calling Web Services from Ajax and jQuery 48](#_Toc490232481)

[RESTful Web Services 50](#_Toc490232482)

[REST: 50](#_Toc490232483)

[OData 50](#_Toc490232484)

[Service Metadata Document 51](#_Toc490232485)

[Data Format 51](#_Toc490232486)

[OData is very case sensitive 53](#_Toc490232487)

[OData Queries 53](#_Toc490232488)

[OData Updates 56](#_Toc490232489)

[Examples 57](#_Toc490232490)

[Module 5: Diving in Deeper! 59](#_Toc490232491)

[Is JavaScript Compiled? 59](#_Toc490232492)

[Data Types and Objects 62](#_Toc490232493)

[JavaScript Primitives 62](#_Toc490232494)

[JavaScript Default Objects 62](#_Toc490232495)

[Window Methods 63](#_Toc490232496)

[Navigator Properties 64](#_Toc490232497)

[JavaScript Objects 64](#_Toc490232498)

[Accessing Object Properties 65](#_Toc490232499)

[JavaScript Numbers 66](#_Toc490232500)

[JavaScript Strings 69](#_Toc490232501)

[Extending Data Types 71](#_Toc490232502)

[Variables 73](#_Toc490232503)

[Scopes 73](#_Toc490232504)

[Hoisting 74](#_Toc490232505)

[Strict Mode 74](#_Toc490232506)

[Creating Variables 75](#_Toc490232507)

[Deleting Variables 75](#_Toc490232508)

[The delete Statement 75](#_Toc490232509)

[Functions 76](#_Toc490232510)

[Function FAQ 76](#_Toc490232511)

[Anonymous Functions 77](#_Toc490232512)

[Closures 77](#_Toc490232513)

[Self-Invoking / Immediately Invoked Functions 79](#_Toc490232514)

[The “this” Keyword 80](#_Toc490232515)

[Errors 82](#_Toc490232516)

[Try Catch Finally Blocks 83](#_Toc490232517)

[The window.onerror Function 83](#_Toc490232518)

[Recursion 84](#_Toc490232519)

[A Factorial Example 84](#_Toc490232520)

# Module 1: Basic Scripting

JavaScript has been described as a simple scripting language, and at the same time described as a complex object oriented language. In this module, we will explore basic scripting. Many of the topics in the module will be expanded upon in later sections of this course.

## JavaScript Basics

In this module we will look at the basics to:

* write and call a JavaScript function,
* find HTML elements in a web page,
* change properties of HTML elements,
* write loops,
* test strings and values (IF statements),
* and a few more…

## JavaScript or ECMAScript?

JavaScript was a creation of Netscape in 1995. Other browser manufactures created their own versions of JavaScript with new, different or incompatible features. ECMAScript is a standard created by Ecma International to standardize the language. ECMAScript has evolved with multiple editions over the years: 1, 2, 3, 5, 5.1, 6 and 7. (There was no 4th edition!) Starting with the 6th edition ECMAScript was renamed using year numbers instead of version numbers. ECMAScript 6 is now called ECMAScript 2015 and the 7th edition is ECMAScript 2016.

* Most modern browsers implement most of ECMAScript 2015 and some of ECMAScript 2016.
* Many browsers have non-ECMAScript features.
* Throughout the web you will find both “ECMAScript” and “JavaScript” used to refer to the language. In this course we will use the more generic “JavaScript” name.

## Which Version of JavaScript am I Using?

JavaScript does not have a built-in version property that you can check. Instead, you will need to test to see if a feature exists. Even if there was a version number available, you could not depend on it as each browser vendor has chosen to implement different subsets and supersets of the language.

Here are a few resources to help sort out which versions are “out there” and which features are supported by which browser.

* https://en.wikipedia.org/wiki/ECMAScript
* https://en.wikipedia.org/wiki/JavaScript#Version\_history
* http://kangax.github.io/compat-table/es5/ and http://kangax.github.io/compat-table/es6/
* http://caniuse.com

To check for support for a feature:

* Lookup the feature here: http://caniuse.com
* Detect at runtime using https://modernizr.com

### Shims, Shivs, and Polyfills

As all browsers have not implemented HTML and CSS to the same level, or even the same way, we will need tools to “level the playing field”. Shims are wedges that you use to level, adjust or tweak things. In the world of HTML, shims are typically CSS and JavaScript that you add to your page to make browsers appear more compatible, let older browsers support new features, and just make the developers life a bit easier. A “shiv” is the same as a “shim”, and the term is most often associated with “HTML5shiv.js”.

A “polyfill” is a shim that implements a modern Application Programming Interface (API) in an older browser without impacting newer browsers that already support the API. If the polyfill is removed from a newer browser that does not need it, it will make no change to functionality. A shim makes changes so all browsers behave the same. As an example, “normalize.css” changes the default margins and removing normalize.css may change the layout of the page. Other Polyfills add support for new HTML5 features in older browsers.

The three terms are often used interchangeably.

For a list of shims, shivs and polyfills see:   
 https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills

## JavaScript

JavaScript is an object-oriented language. As with most programming languages, JavaScript has no user interface and must be used within a host application. In this course, we will only consider one host application, the web browser. The top-level browser object seen by JavaScript is named “window”. All other objects, including core JavaScript objects, are properties of this object. In tabbed browsers, each tab has its own window object.

**JavaScript:**

* **Is not Java!** While both share many constructs with the rest of the C family of languages, it is not derived from Java.
* **Is object oriented.** For that matter, everything in JavaScript is an object! But JavaScript does not use classes, class instances or class-style inheritance.
* **Is not a standalone application.** You cannot create a JavaScript console application to run from the desktop. JavaScript is embedded into a host applications, and for our discussions, JavaScript is embedded in a web browser.
* **Has no user interface.** Any interaction with a user will be through the host. In the case of a browser this the browser’s debugging console, a web page or features like the alert() popup.

## Writing and Testing JavaScript

### Editors

While you can write JavaScript using just a text editor like Notepad, you will get the best results from a dedicated HTML / JavaScript editor.

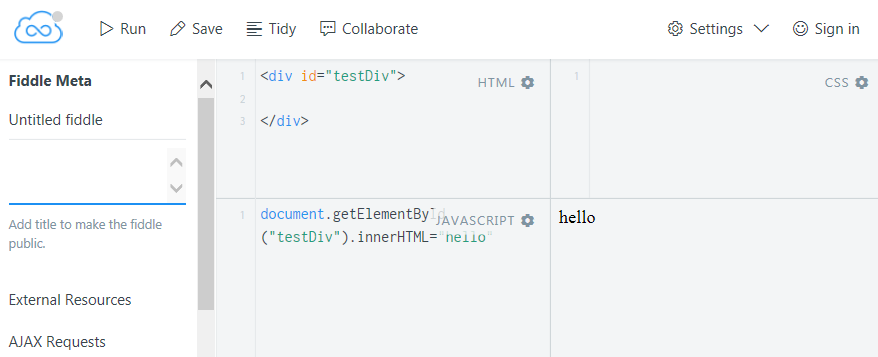
Common HTML / JavaScript editor features:

* Keyword highlighting using colors.
* Auto indenting for nested HTML tags or JavaScript code.
* IntelliSense and autocomplete to validate code as you type.
* Snippets to automatically write common code structures for you.
* Debugging integration with popular browsers.

Examples of HTML / CSS editors.

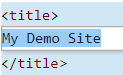
* Aptana Studio
* Adobe Dreamweaver
* Visual Studio
* Visual Studio Code

Tools to test and debug JavaScript:

* Browser built-in F12 developer tools.
* Browser and editor integration such as Visual Studio and Internet Explorer.
* Online tools like:
  + W3Schools JavaScript editor: https://www.w3schools.com/JS/tryit.asp?filename=tryjs\_events
  + Quackit: http://www.quackit.com/html/online-html-editor/
  + HTML-Online: https://html-online.com/editor/
  + JS Fiddle: https://jsfiddle.net/
    - You can do live testing of HTML, CSS and JavaScript using JS Fiddle.
    - You can save and share examples of your code. (Here’s the link to the example displayed below: https://jsfiddle.net/emjdpxdo/1/)  
      
  + JS.do: http://js.do/

### A Demonstration of Using the Browser’s F12 Tools

Note: The steps below are for Internet Explorer 11, but are similar for other browsers.

1. Open Internet Explorer (or Microsoft Edge).
2. Press F12 or click Tools and F12 Developer Tools.
3. Click the Console tab.
4. At the console prompt at the bottom of the window, enter some JavaScript to test…
   1. Example: alert("Hello!")
   2. And press Enter.
5. Visit a web site and interrogate the Document Object Model of the page:
   1. Visit a web site such as www.wikipedia.org.
   2. Examples to try into the Console tab: (Type the example and press Enter.)
      1. document.title // displays the <title> text
      2. document.title = "Hello world" // changes the <title> text (but only in the in memory version)
      3. document.getElementsByTagName("table") // displays the collection of tables  
          <HtmlCollection length="0"></HtmlCollection> (0 = no tables found!)
      4. document.getElementsByTagName("div") // displays the collection of divs.  
          <HtmlCollection length="51">...</HtmlCollection>   
          (51 divs in this example)
6. Explore the DOM:
   1. Click the DOM Explorer tab. (The Inspector tab in Firefox and the Elements tab in Chrome.)
   2. Expand the <head> element.  
      
   3. Find the <title> element.  
      
   4. Change the contents of the <title> element.
      1. Double-click the text, enter the new text and press Enter.  
         
      2. Confirm that the text in the browser’s tab has changed.

Notes:

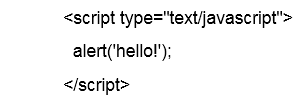
* Changes made using JavaScript code or the editing of the HTML and CSS in the F12 tools will not change the original page. Refreshing the browser will restore the original page.
* While you can edit element values and attributes, you cannot write new HTML code from the DOM Explorer.

## Key Parts of a Script

#### Basic JavaScript Syntax

JavaScript is written inside of <script> tags embedded in tag attributes, called from HTML element events (onclick) or loaded from linked text files.

This example displays a popup message from a script block:



Note: The “type” attribute is optional in HTML 5 and required in earlier versions.

##### JavaScript can also be directly called from element events.

For example, "onclick" is the event raised when a use clicks a button.



##### JavaScript can be loaded from files.

The <script> tag can also be used to load JavaScript from a file.

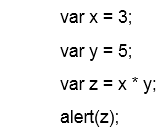


##### <script> tag notes:

* If the src attribute is present, then the content (text between the start and end tags) must be empty. <script src="…”></script>
* Even though when using the src attribute there is no content between the <script> tags, this tag cannot be self closed (/>) and must have an end tag.
* By default, <script> tags are “blocking” and will stop the loading of page content until the entire script file has been downloaded.
  + In some cases, placing these script tags at the end of the page (i.e. just before </body>) is a good practice. All other content will be loaded before the JavaScript is loaded.
  + <script> tags that load external files (i.e. include “src=”) can also include the “defer” attribute. This will start downloads of these files in parallel to other page content, and defer their execution until just before the DOMContentLoaded event fires. (DOMContentLoaded fires just after all of the HTML has been downloaded and parsed. It does not wait for stylesheets, images or subframes to finish loading.)
  + <script> tags that load external files (i.e. include “src=”) can also include the “async” attribute. The script will be executed as soon as it is loaded and will run in parallel with the rest of the page is still loading.
* The <noscript> block can be used to display a message to users who have turned off JavaScript in their browser.  
   
* The type attribute is optional with HTML5, but required for older versions. If omitted, it will default to “text/JavaScript”. “Required” means the specification says it is required. Most browsers do not require it.

#### JavaScript is written as a series of statements separated by semicolons.

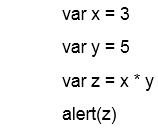
Four statements, written on four lines.



The following is identical to the above code. Four statements on one line.

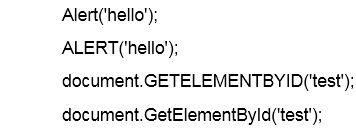


The semicolons are often omitted when each statement is on its own line.

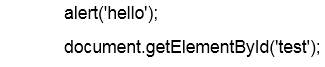


#### JavaScript is "case sensitive"

These examples will fail:



These examples will work:



Notes:

* JavaScript is very case sensitive. The following are all different variables: “bankaccount”, “BankAccount”, “BANKACCOUNT”.
* With the exception of the Math and JSON object, most of the keywords and methods in JavaScript start with a lowercase letter.
* In general, function libraries like Math and JSON use capital letters.

#### Comments

The two toughest coding things you will have to do as a developer is figure out someone else's code, or your own code six months after you have written it. Document your work! Add comments to explain what you did, or where you borrowed the code from.

Any text that follows two slashes (“//”) is ignored. Any text between /\*” and “\*/” is ignored.

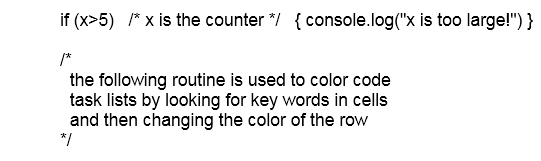
Add a comment as a single line:



Add a comment to the end of a line:



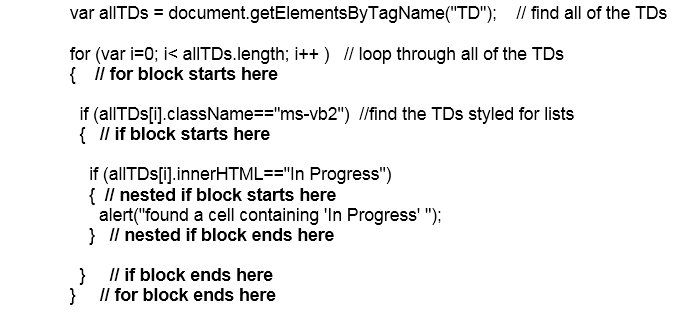
Add a block of comments between "/\*" and "\*/":



#### Code Blocks

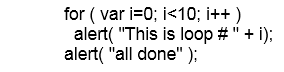
JavaScript structures work with blocks of code that are enclosed within curly brackets. Many of the JavaScript blocks that you will find in this course are for IF, FOR and FUNCTION statements. Code blocks are frequently nested inside of other code blocks.

Here's an example of using a FOR loop to repeat a block of code for each TD element found in the page. It includes several nested code blocks.

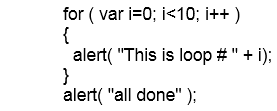


While the curly brackets are required when the block has more than one line of code, they are optional if the block has only one line. *Your code will be easier to understand if you always add the brackets*.

This works without the brackets.



But this is easier to understand as the brackets show which lines of code that are used with the FOR.



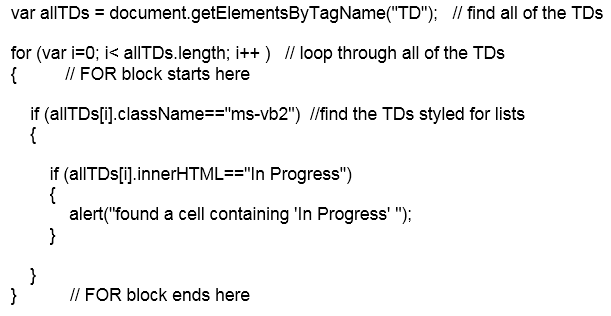
#### Spaces, line breaks and semicolons

While JavaScript generally does not need line breaks, indents or extra spaces, these will make the code much easier for humans to read. Semicolons are the "official" end of line punctuation, but code will still work if the line of code ends with a line break. Semicolons are not needed after closing curly brackets ( } ).

The following works as one long line of code, but is very hard to read and debug:



This code is much easier to read when nicely formatted, and you can easily see if you are missing a bracket somewhere:

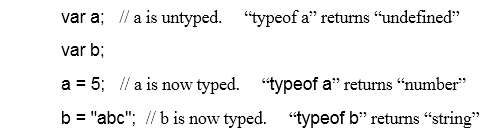


Many JavaScript libraries do look like the first example. By removing the excess characters and comments you can shrink the file size and reduce download time, but at the expense of readability. If you do this, keep in mind that end of line semicolons are required, and that while block comments ( /\* … \*/ ) will work, double slash ("//") comments will not.

#### JavaScript Variables

*Note: This is a very brief introduction to variables. They are covered in depth in a later module.*

Variables are not assigned data types when they are created as in C# or VB.NET.



Multiple variables can be declared in a single statement;

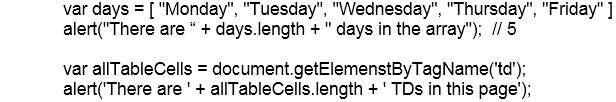


Variables can store simple data types such as strings and numbers. A variable can also store an object such as an HTML element (tag). This example finds a table with an ID of "table1" and stores it in a variable named "atable" and then changes a property of that table.

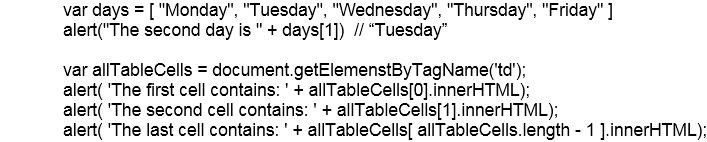


#### Arrays

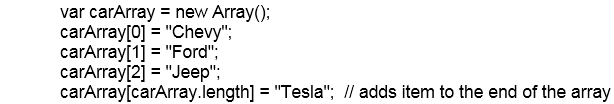
Variables can store arrays, including arrays of HTML elements. Arrays can be initialized by listing array elements between square brackets or from a collection source like document.getElementsByTagName.



Individual array elements are addressed by using an indexer ( [*index*] ) that starts with zero. An array of ten elements would have items with indexes from 0 to 9.



Arrays can be expanded in code by adding new items to the end of the array. The following creates an empty array and then adds three items.



Arrays can also be expanded using the push method.



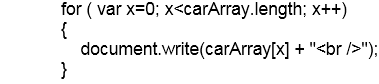
The last element of an array can be retrieved and then deleted from the array using pop. You can use push and pop to create a stack (first in, last out) structure.



Arrays can also be created by defining a list of items between square brackets:



Arrays are often accessed using a FOR loop:

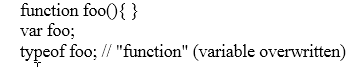


#### JavaScript Functions

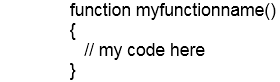
*Note: This is a very brief introduction to functions. They are covered in depth in Module 5.*

Functions are "reusable blocks of code" that you can write once and then call one or many times from other code. Functions are declared with the “function” keyword, include zero to many parameters and a code block enclosed by curly brackets ( {…} ).

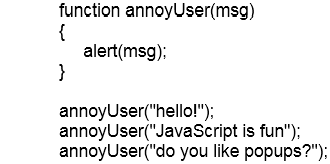
**Notes:**

* **Functions are stored as variables.** If a script has a variable named “foo” and a function named “foo” then the variable will always be overwritten by the function, regardless of the order in the code.
  + This:  
    
  + is the same as this:  
    
* **Functions are created at the Global scope by default.** (I.e. they are a property of the window object. window.foo()
* **Functions can be declared inside of another function.** This hides them from the Global scope and helps avoid naming conflicts.

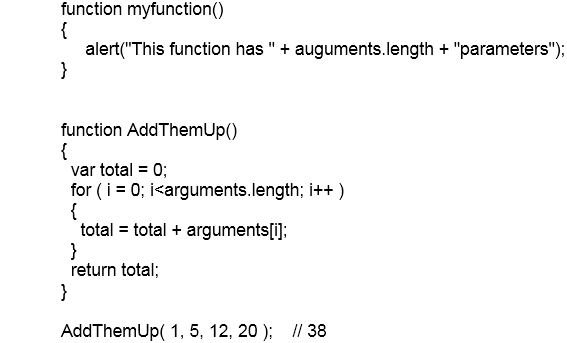
A function declaration can be as simple as:



Functions can also have parameters so you can pass different data to the function each time it is called.



Instead of listing the parameter names in the function declaration, you can use the “arguments” object to accept any number of parameters.



#### Conditional Statements

JavaScript includes the common set of conditional statements found in the C family of languages: if, switch, for, do and while.

##### Testing values - IF

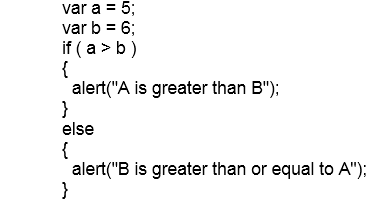
JavaScript's IF statement is used to conditionally run a block of code. The most common mistake made by new JavaScript developers is typing just one equal sign when doing comparisons.

Example - if A equals B then display an alert:

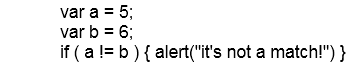


Note in this example the use of the single equal sign for assignments ( a = 5 ) and the double equal signs for equality (a == b).

Example - if A is greater than B then display an alert else display a different alert:

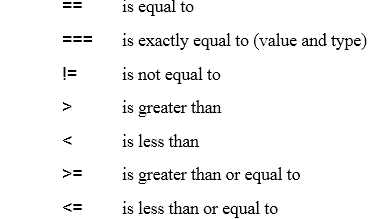


Example - if A and B are not equal then display an alert:



##### JavaScript Comparison Operators

The JavaScript comparison operators are similar to those found the C family of languages.



The “===” comparison operator is similar to the following that tests both the values and the datatypes:



##### The Conditional “ternary” Operator.

Note: “ternary” = “composed of three items”.

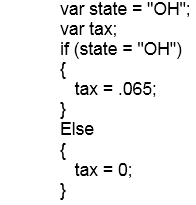
JavaScript includes an operator “?” that behaves as an IF structure.



Example:

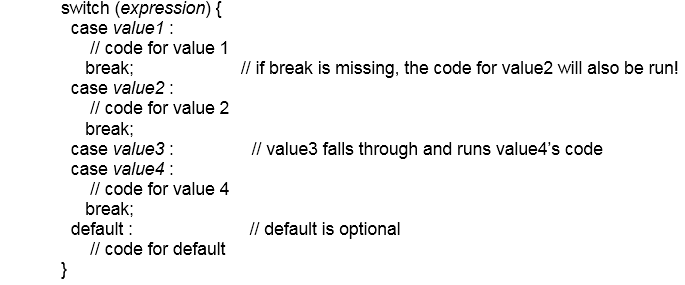


The above is equivalent to:



##### Switch

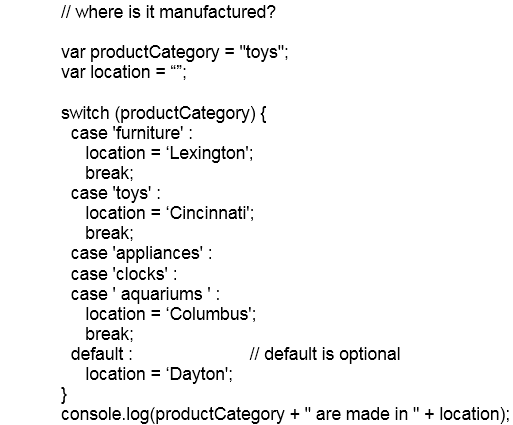
The switch statement accepts an expression or value and uses that value to select a case clause. The switch statement is often described as the equivalent of nested IF statements.



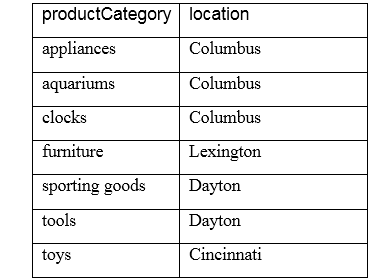
Notes:

* Except where there is a missing break statement, case and default statement sections can be listed in any order.
* Code blocks can have many lines of code, including other nested statements.

Example:



From the above:



#### Looping Statements

JavaScript supplies several statements for creating code that loops. Loops are blocks of code that run while, or until, a certain condition is true.

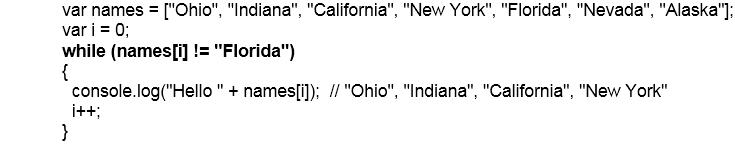
* **“for” loops** use a counter and run for a known number of iterations.
* **“while” loops** run zero to many times and exit when an expression changes to false. The expression is evaluated at the start of each loop.
* **“do”** loops run one to many times and exit when an expression changes to false. The expression is evaluated at the end of each loop.

Notes:

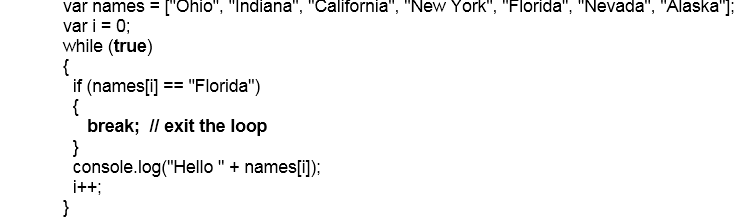
* Something inside of the loop must change the expression, otherwise you will have created an endless loop.
  + while (true) { /\* do this forever \*/ } // endless loop, unless there is a break statement.
* Loops can exit at any time by using break.
  + while (true) { if (someexpression) { break }; /\* do stuff \*/ }
* Loops inside of a function can exit at any time by using return to exit the function.
* Loops can skip to the next iteration by using the continue statement.

##### WHILE

The while loop tests an expression before executing a block of code and repeats the block while the expression equals true.

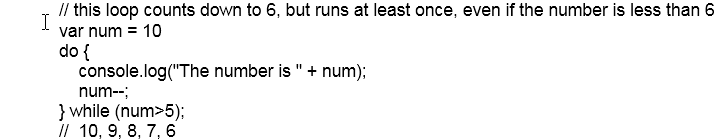


The following example loop would run forever if not for the break statement. This example produces the same results as above.

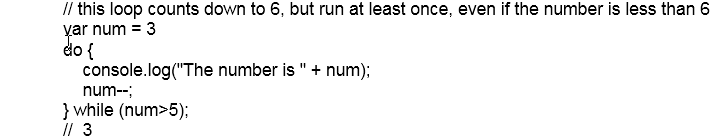


##### DO WHILE

A do loop is similar to a while loop, but tests the expression at the end of each loop. A do loop will always run at least once.



Even with num=3 this will still display at least one message.



##### FOR

A for loop “counts”. It needs three parameters to define a counter variable, a test for completion, and code to increment or decrement the counter.



The counter is typically initialized in the for statement, but could also be a function or globally scoped variable.

Count from 5 to 9:



Count from 5 to 9:



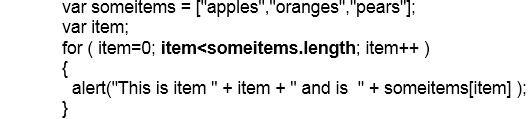
In the next example, the “var” may lead you to believe that the variable is scoped to the FOR block. It is not! It is “hoisted” to the function that contains the FOR loop, or to the Global scope.



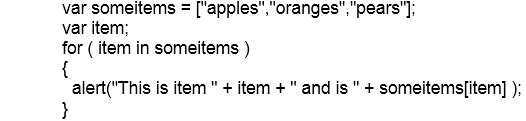
##### Looping through an array

Many of the DOM selectors such as getElementsByTagName return an array of HTML elements. Once you have the HTML elements stored in an array, you can then loop through them with a "FOR" loop.

If your FOR loop always loops through all the items in an array then you can use this version:



Or you could use this shorter FOR - IN version:

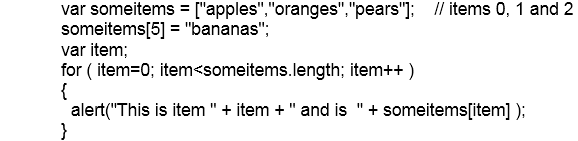


Notes:

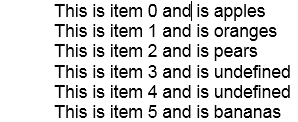
* In both of the above examples, the “item” variable is an integer that represents the “nth” item.
* Array indexes in JavaScript are numbered starting with zero!
* FOR…IN is similar to FOR…EACH found in other languages. JavaScript even has a FOR EACH, but it is deprecated in favor of the FOR…IN. A simple FOR is preferred over FOR…IN when working with arrays. (Details in below…)

##### FOR Loops and Sparse Arrays

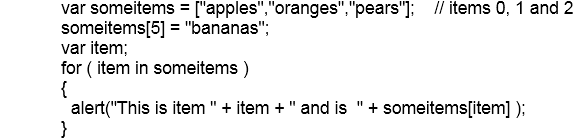
The difference between FOR and FOR…IN is in how they handle “sparse arrays”. A sparse array is missing values. For example, the following code creates array elements with indexers of 0, 1, 2, and 5. Elements 3 and 4 have never been defined.



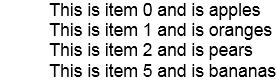
The above returns:



Modifying this code to use FOR…IN only returns four items.



The above returns:



##### For, Blocks and Scope

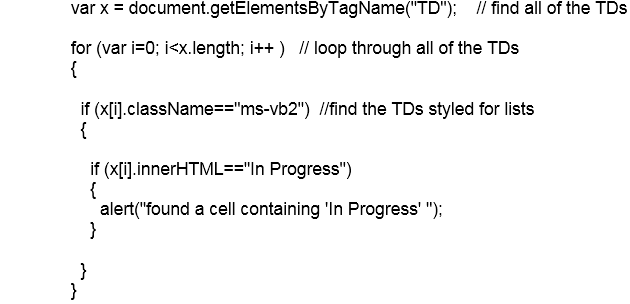
We have mentioned blocks before, but now is a good time for a review. Note that FOR is a block structure and that the start and the end of the block is marked with curly brackets. The brackets are optional if there is only one line of code in the block. The following repeats only the next statement in the code.



JavaScript does not care about line breaks, so both of the following are also identical to the previous example:

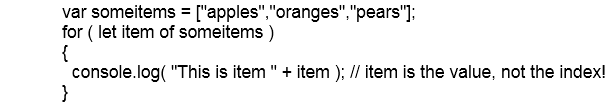


Blank lines, spaces and tabs are free! Use them to improve readability. Here's a commented and nicely formatted example:



##### FOR OF

ECMAScript 2015 (6) adds a FOR OF loop that is similar to the FOR EACH found in other languages. At this time, FOR OF does not work in Internet Explorer 11, but does work in other modern browsers.



See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of

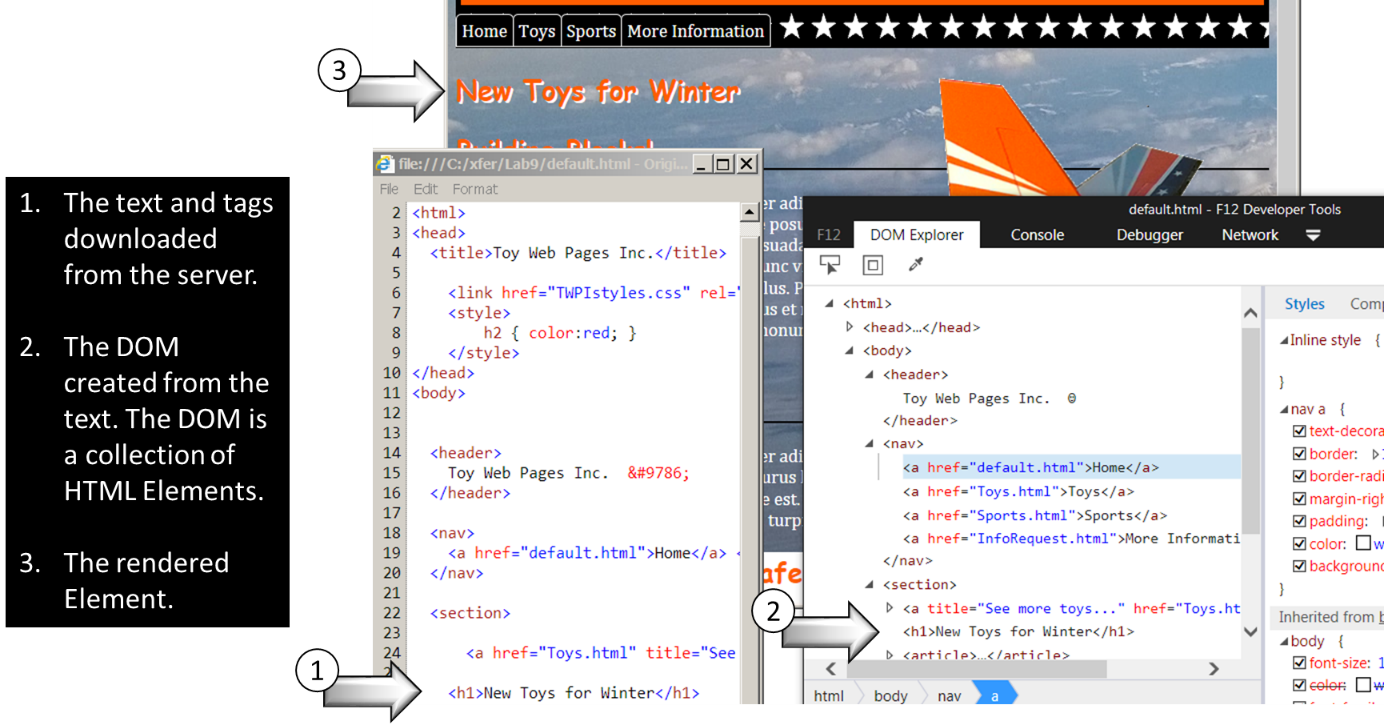
## Useless Trivia!

* JavaScript was first called "Mocha", and when first released with Netscape 2.0 in September 1995 JavaScript was called "LiveScript".
* LiveScript was renamed to JavaScript in December of 1995 as part of a partnership with Sun to integrate Java applets into LiveScript and Netscape.
* Microsoft added JavaScript to Internet Explorer 3.0, but called it Jscript.
* In November of 1996 JavaScript became formalized into an international standard named ECMAScript.

# Module 2: The Document Object Model

## The DOM

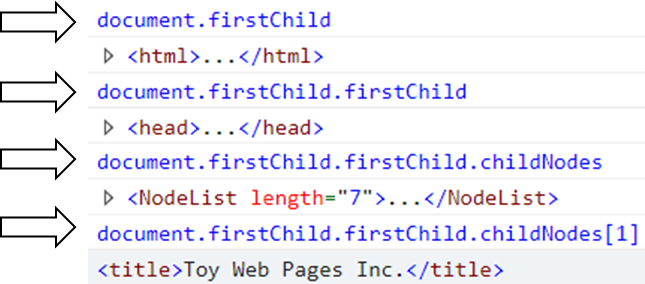
Neither browsers nor JavaScript work directly with the HTML file that was received from the web server. The browser parses this file and converts it into a tree structure called the Document Object Model, or DOM for short. The DOM is an in-memory collection of objects that represents the elements (tags) defined in the source HTML file, or created by JavaScript.



## Accessing Objects from the DOM

After the HTML has been parsed into the DOM, the individual elements are stored as objects in a tree structure. This structure is the window.document object, which is almost always shortened in code to just document.

The document object can be accessed by starting at the top, and then drilling down through the collection of child nodes. While this can be done, there are better ways!



### Document Properties

When the DOM is created, browsers will pull out some of the content and make it available as properties of the document object.

Built-in properties for frequently accessed document objects.

* document.anchors – the collection of <a> tags that have “name” attribute. (Not too useful anymore as HTML5 does not support the name attribute on <a> tags!)
* document.links – the collection of <a> tags. I.e. all links on a page.
* document.head – the <head> element - the <head> section of a document.
* document.body – the <body> element - the <body> section of a document.
* document.images – the collection of images.
* docment.url – the document location.

For a complete list see: https://developer.mozilla.org/en-US/docs/Web/API/Document  
or https://www.w3schools.com/jsref/dom\_obj\_document.asp

### Document Methods to Find Elements

Most of your JavaScript work will probably be searching and updating the DOM. The document object includes a number of methods to make it easy to find and work with page elements (tags).

These two examples show working with the entire document or with the children of a single object.

* var tables = document.getElementsByTagName("table"); // all tables in the document
* var mytable = document.getElementById("MyTable");  
  var rows = mytable.getElementsByTagName("tr"); // all rows in this one table

#### Searching for Elements

These methods search by a single element characteristic: ID, Class Name, Name and Tag Name.

* getElementById() – Returns the single element with the specified ID. If no element found, this returns null.
  + Note! The text of the ID is case sensitive! getElementById("abc") will not find id="Abc".
* getElementsByClassName() – Returns a collection of elements that have the specified class name. If no element found, this returns an empty collection. (I.e. not a null.)
* getElementsByName() – Returns a collection of form elements that have the specified name attribute value. If no element found, this returns an empty collection. (I.e. not a null.)
* getElementsByTagName() – Returns a collection of elements that have the specified tag name. If no element found, this returns an empty collection. (I.e. not a null.)

Examples:

* var tables = document.**getElementsByTagName**("table");  
  console.log(tables.length) // 0 .. many
* var images = document.**getElementsByTagName**("img");  
  console.log(tables.length) // 0 .. many
* <div class="lefttoright **nav** bluetext">…<div>  
  var navdiv = document.**getElementsByClassName**("**nav**");   
  console.log(tables.length) // 0 .. many
* <div id="**corpLogo**"> … </div>  
  var navdiv = document.**getElementsById**("**corpLogo** ");   
  // navdiv = null or a DIV object

#### Searching for Elements using CSS Selectors

CSS uses selectors to find elements by ID, tag name or class. In modern browsers, you can use .querySelector and .querySelectorAll to find elements using CSS-like selectors. These selectors range from simple select by ID selectors ("#someId") to complex queries like "div.upcoming-events-section div.row:first-child".

* querySelector() – Returns the first element that matches a CSS style selector.
* querySelectorAll() – Returns a collection of elements that match a CSS style selector.

Select for tags:

* var tables = document.**querySelectorAll**("table");  
  console.log(tables.length) // 0 .. many
* var tables = document.**querySelector**("table");  
  console.log(tables.length) // null or a TABLE object

Search for complex patterns:

* var tables = document.**querySelector**("div.upcoming-events-section div.row:first-child");  
  console.log(tables.length) // null or a single element object

Search for elements with a specified attribute:

* var inputtexttags = document.querySelectorAll("**input[type='text']**")

For browser compatibility see:

* https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector#Browser\_Compatibility
* http://caniuse.com/#feat=queryselector

For more on CSS Selectors see:

* https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction\_to\_CSS/Selectors
* https://www.w3schools.com/cssref/css\_selectors.asp
* http://www.w3.org/TR/CSS2/selector.html

## Responding to Events

Most HTML elements raise events in response to user or external actions. You can write JavaScript code to respond to these events.

Examples:

* The <body> tag has an “onload” event to signal that the page has been fully loaded.
* The <button> tag has an “onclick” event to signal that a user has clicked the button, or that the button’s “click” method has been called.
* The <div> tag has an “ondrag” event to process the start of a drag and drop operation.

### Common Events

The following are some of the more common events:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Mouse events** | **Keyboard events** | **Drag/Drop events** | **Form events** | **And more!** |
| onclick | Onkeydown | ondrag | onsubmit | oncopy |
| ondoubleclick | Onkeyup | ondrop | onreset | onpaste |
| onmouseover | onkeypress | ondragenter | onchange | oncut |
| onmouseout |  | ondragleave | oninput | ononline |
|  |  | ondragstart | oninvalid | onoffline |
|  |  | ondragend |  | onwheel |

Also see:

* https://www.w3schools.com/jsref/dom\_obj\_event.asp
* https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers

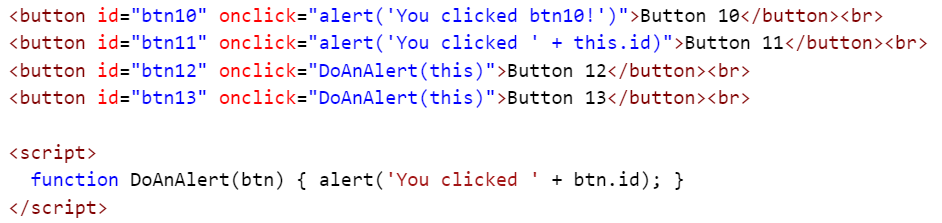
### Adding Event Handler Code

JavaScript code can be added directly to an element’s event attribute or added to the element by code.

Sample File: Module2\Demos\events.html

#### Adding an event handler in an attribute.

You can add JavaScript code directly to the elements event attribute using either inline JavaScript code or a function call. The “this” object is useful for passing the element object to the function.

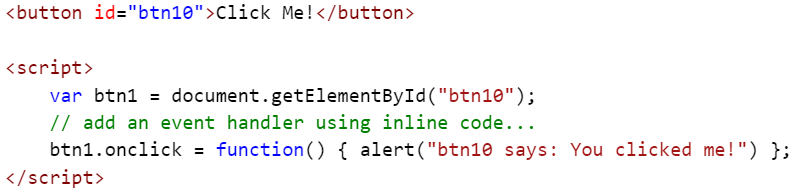


#### Adding an event handler from code.

Embedding JavaScript in HTML tags is not always the best practice. You can programmatically add event handlers at run time so you can separate design (HTML) from code (JavaScript), or to assign event handlers based on an algorithm.

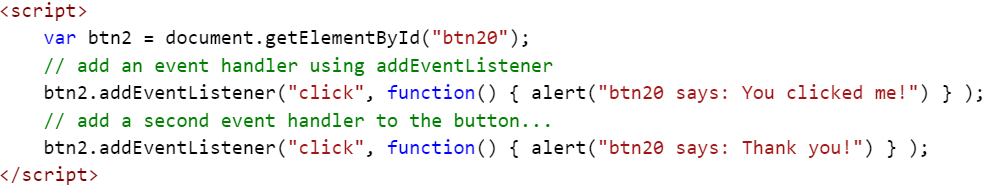
Sample file: Module2\Demos\AddEventHandler.html

**Option 1:** Directly assign a function to the event property. (“onclick” in this example)



**Option 2:** Use the addEventListener function.

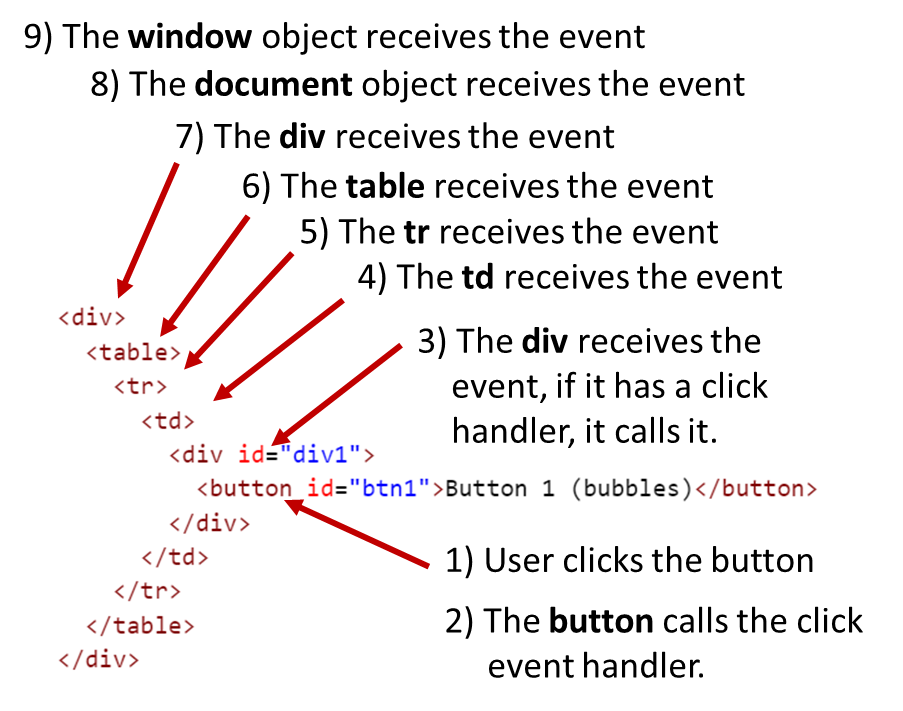
addEventListener has a major advantage over the previous example: You can add more than one event handler to a single event.



## Event Bubbling / Event Propagation

By default, an event will be passed up the chain of elements, firing event handlers as it goes, until it reaches the window object. In the example below we have a <button> contained in a <div>. By default, clicking the <button> will also fire the click event for the <div>. The click was on the button, but then bubbled up to the parent <div>, and then bubbled up to that <div>’s parent, all the way to the top of the DOM.

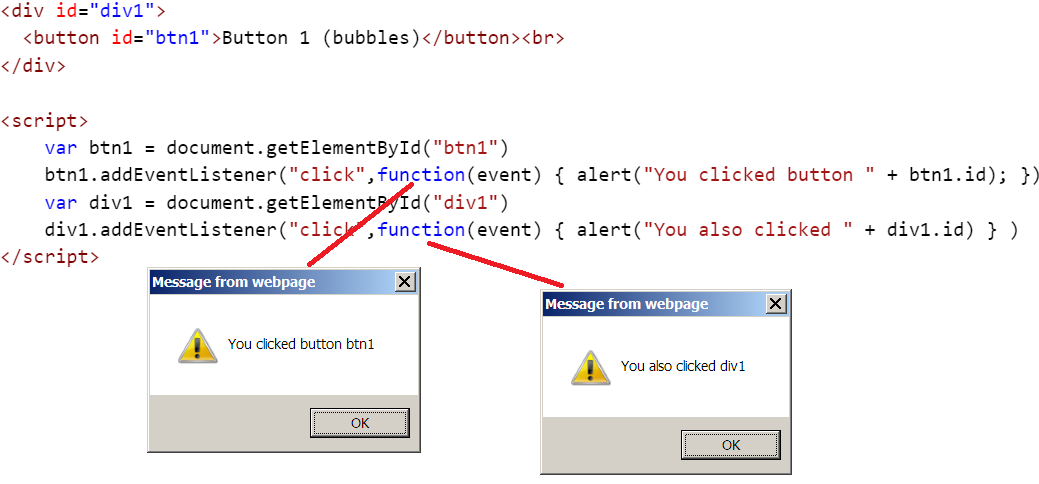
Sample File: Module2\Demos\BubbleDemo.html



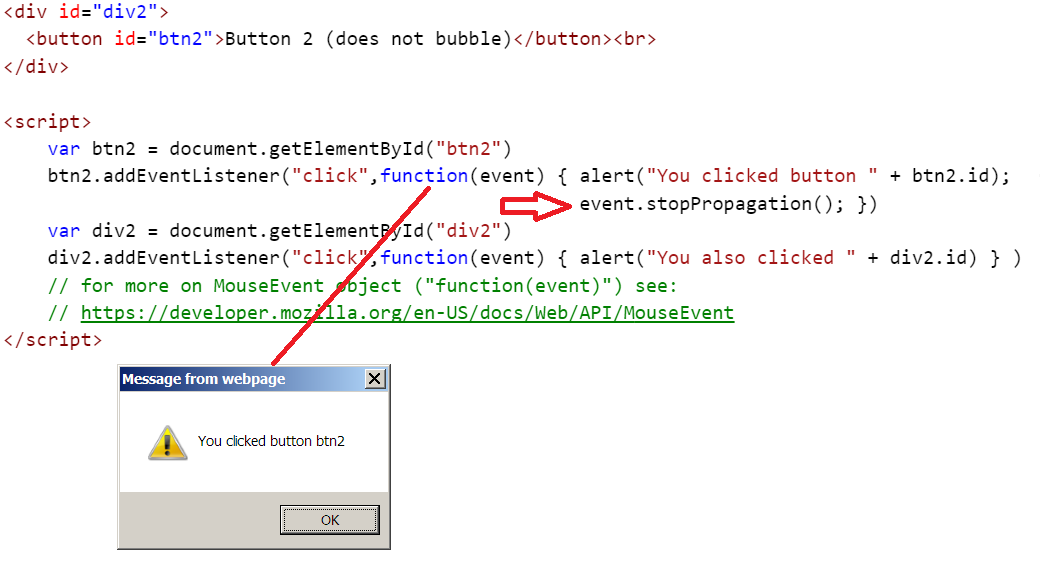
### Stopping Event Propagation

In the following example, a single button click raises the button click event and then the div click event.

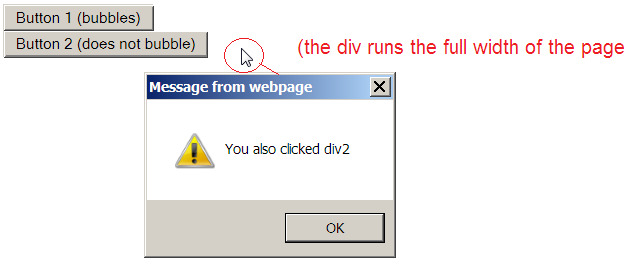
Sample File: Module2\Demos\Bubble.html



To prevent bubbling / propagation, add a parameter the event handler function to capture the event object, then call event.stopPropagation. In this next example the event on the div is never triggered from the button click.



The event handler on the div is still “alive and listening” and will be fired if you click the div somewhere outside of the button.



## Adding Elements to the DOM

JavaScript can add new elements to a page to dynamically create new content and structure. Elements can be added from strings of HTML text or by creating new element objects.

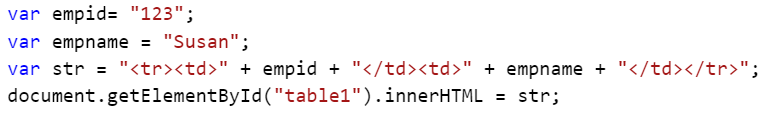
Sample File: Module2\Demos\AddingElements.html

### Create HTML from Strings

You can use the innerHTML property to add HTML elements to a page. Although created as strings, these are added to the DOM as regular elements.



…

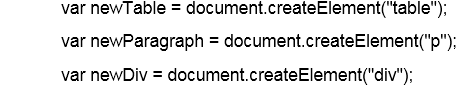




### Creating New Elements

Elements can be created as new objects and then added to the DOM. New elements are created using the document object’s createElement method. Adding a new element has three basic steps: create the new element, set any properties, and add to an existing element.

Creating a new element object in memory.



After creating the element you will typically add attributes. (more on this later)

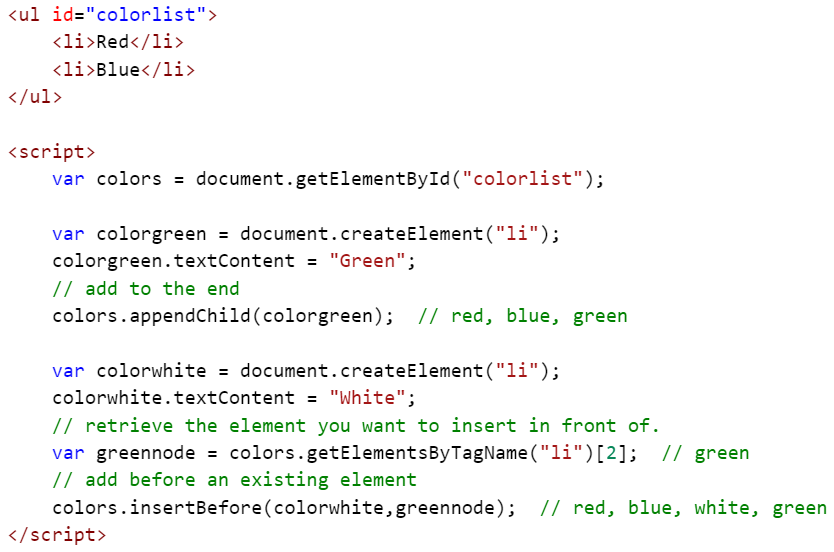


The new element will then be added to the DOM by one of several methods.

* Add the new element as the last child of an element.  
  
* Add the new element before an existing element.  
  

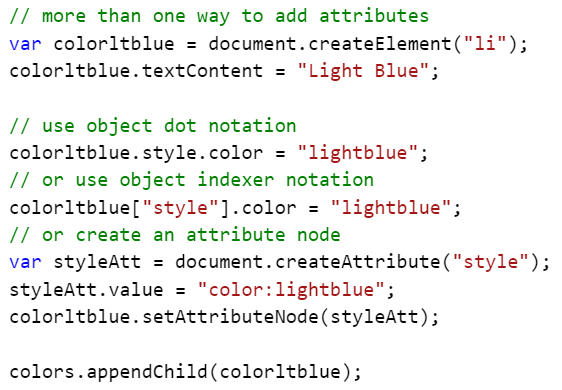
Example:

Sample File: Module2\Demos\AddingElements2.html



### Adding Attributes to New Elements

When creating new elements, or updating existing elements, you will need to set attribute values. The DOM gives us at least three ways of doing this. Using the object dot notation style is a bit more object oriented, and the validation features of an editor verify your attribute property names.

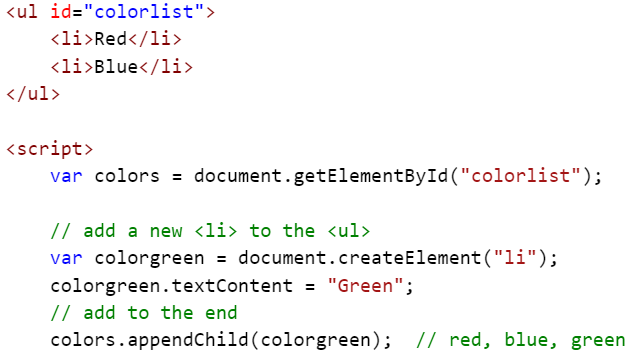


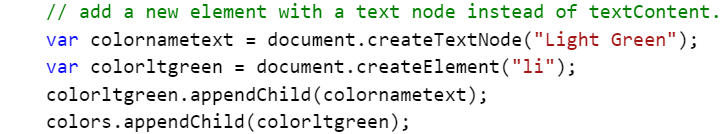
### Adding Text Inside of a New Element

Text can be added to a new element either by setting the element’s textContent property, or by creating a new Text Node using createTextNode and adding it as a child of the new element. While the textContent property is faster and simpler, the use of createTextNode will create an object that can be inserted before or after other child objects.

* 
* 

Example:



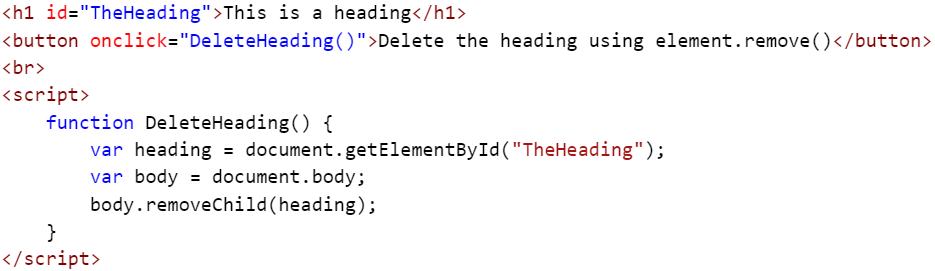


### Removing Elements from the DOM

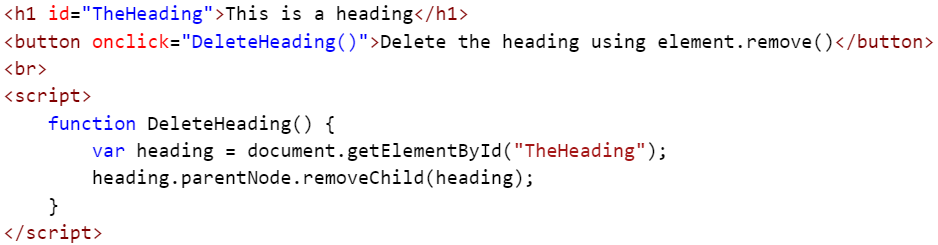
Elements can be removed by removing them from their parent. This process requires the retrieval of two objects, the child to remove, and the parent of the child.



As a shortcut you can use the element’s parentNode property to discover its parent. In the example below, the parent of the <h1> element is the <body> element. Instead of retrieving the body element we simply get the parent (the body) from the <h1> element and then call removeChild.



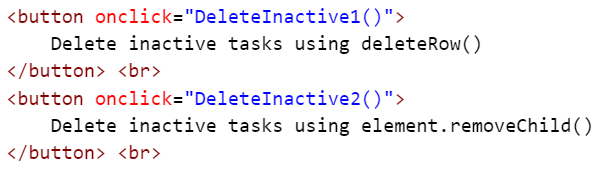
Using the “parentNote” shortcut.

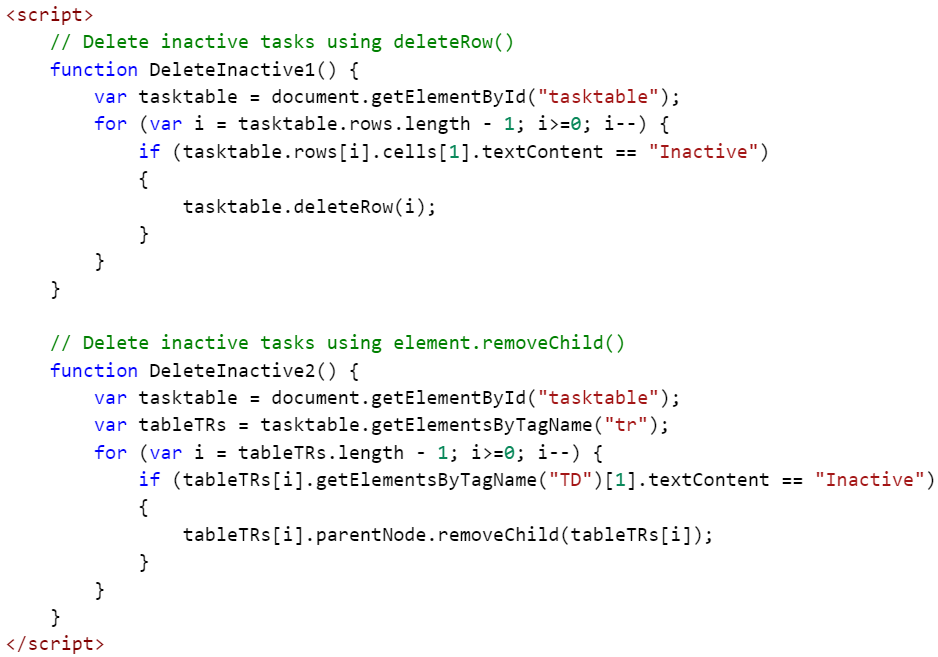


Some elements have their own “remove” methods. As an example, a table object includes methods to add and delete entire rows.









### Delete in Reverse Order

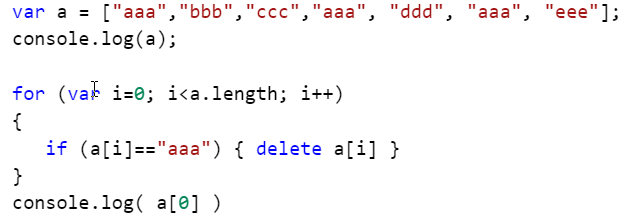
Note the “count down” loop in the examples above. When removing items from a collection, start at the end.

JavaScript offers several ways to remove items from an array:

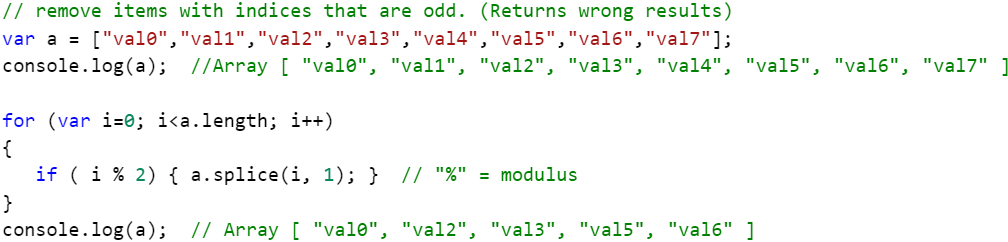
* arrayvar.pop() – Removes the last item in the array.
* delete arrayvar(*index*) – Sets the item to “undefined”, leaving the array as a sparse array.
* array.splice(*index*, *qty*) – Removes items starting at the specified index. “splice” can also be used to insert items into an array. See https://www.w3schools.com/jsref/jsref\_splice.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array/splice
* You can also use a “move” algorithm to move items down to overwrite the one you want to remove.

Sample File: Module5\Demos\DeletingItems.html

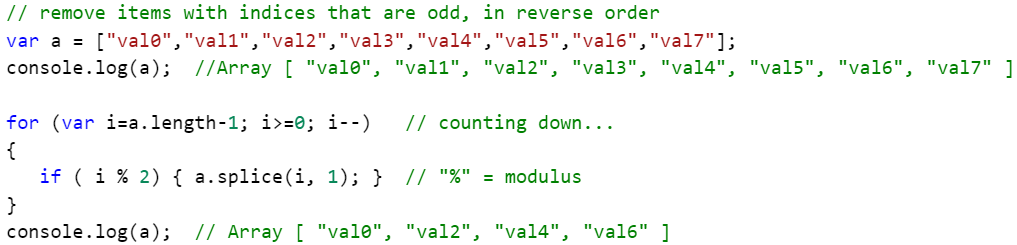
**Using delete** (Leaves empty nodes. Value = undefined.)



**Using splice** (Removes item from the array, but the array indices can get “confused”. The first example below returns incorrect data!

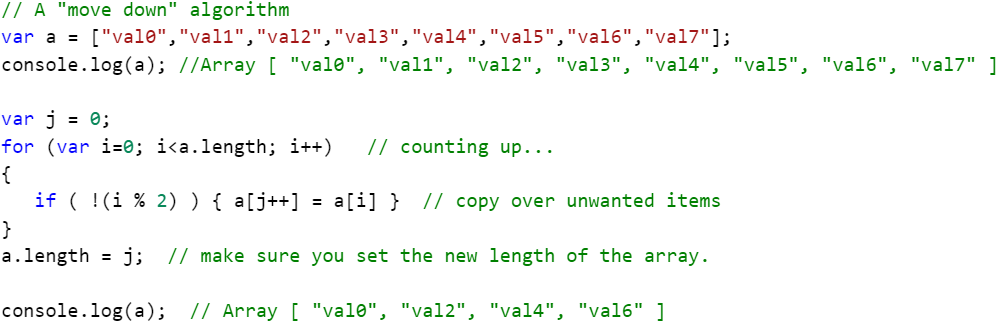


When actually removing (not setting to “undefined”) items in an array you will need to work backwards through the array elements. Here’s the same example as above, but it’s looping through the array in reverse order and returning the correct results. The change to the code is in the “for” line.



**A “Move Down” Routine**

This routine overwrites unwanted items and then resets the length property.



# Module 3: Introduction to jQuery

## jQuery

A lot of the coding work you will do with JavaScript will be to first find an HTML element, and then manipulate that element. Often, all we are doing is some searching, or querying, of the HTML and then performing a quick update. jQuery is ideal for this.

jQuery:

* Is a library of JavaScript code.
* Is a single JavaScript function named “jQuery” and is most often accessed with an alias of “$”.
* Is primarily used to “query” the DOM to find elements, and then perform an action against them.
* Includes a library of helper functions such as “$.ajax()”.

### First the negatives:

* jQuery is not part of JavaScript or the browser. You will either download the jQuery library from the web and upload it to your server, or you will directly link to a copy of the library on the web.
* The library is not large, but it will add to page load time, at least the first time a user visits a page. After the first visit the user's browser will use a local cached copy of the library.
* The names of common operations (functions etc.) are not the same as JavaScript.
* It’s something else to learn. ☺

### And then the positives:

* jQuery can create some very compact solutions, often with just one line of code.
* jQuery is just a single function call into a library, so it can be intermixed with regular JavaScript. In fact, all of jQuery is just one big JavaScript function named jQuery() or $().
* There are a lot of web resources for jQuery.
* jQuery is extensible. There are many add-in libraries available.

What follows is a very, very, brief overview of jQuery. For more on jQuery see:

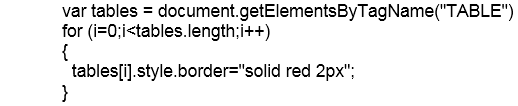
* jquery.com - the official web site for jQuery
* "jQuery: Novice to Ninja" by Earle Castledine and Craig Sharkie

### JavaScript vs. jQuery

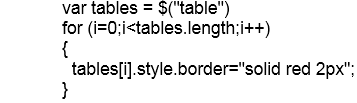
jQuery is written in JavaScript, so everything you can do with jQuery can be done using plain ole JavaScript. jQuery typically adds brevity (one or two lines do the work of many JavaScript lines), clarity (you don’t need to study blocks of “if” or “while” logic) and cleaner code (you are not creating lots of variables and little functions).

Consider how you would discover and highlight all of the tables in a page. You could write a JavaScript routine to loop through all of the tables, then turn on borders and then set the width and color.

Here’s a JavaScript only way to style each table. Two variables are created.



Here’s a jQuery JavaScript hybrid. jQuery returns an array of tables, and normal JavaScript styles each table. We are using jQuery ($) with a CSS style selector to find the table elements. The code is a bit more brief and the CSS style selectors let you avoid a lot of code to select just the right elements. Two variables are still created.



Here's a “pure” jQuery example to make the same style change. One line of code, no logic to study and no extra variables created.



## Take a Peek…

If you would like to take a look at the jQuery library code (only 10,000 lines or so):

* Using Firefox, Chrome, Edge: Navigate to https://code.jquery.com/jquery-3.2.1.js and the code will be displayed in the browser window.
* Using IE 11 and other browsers: Navigate to https://code.jquery.com/jquery-3.2.1.js, download the file and then open it in Notepad.

The URL above is for the “development” version of the library that includes nice formatting, meaningful variable names and comments. This one is very useful when you are debugging your jQuery code. In your production site, you will probably use the compressed or “minimized” version of the library. Repeat the above steps and take a look at https://code.jquery.com/jquery-3.2.1.min.js. It’s now just one long, very long, line of JavaScript code. (Your text editor or browser may display it as several lines.)

The development version of jQuery 3.2.1 is 262 kb while the compressed version is only 85 kb.

## Downloading the jQuery Library

Downloading your own copy of the library and hosting it on your web servers is often the preferred method of accessing the library, especially if you are not supporting internet users. You will then know the library is where you need it, will still be there when you need it, and that it will not change.

### jQuery Versions

For a long time, jQuery had two parallel versions, the 1 series and the 2 series. The 2 series is faster and smaller, but does not support Internet Explorer before version 9.

In 2016 jQuery 3.0 was released. Like the 2 series, it does not support IE before version 9. While very compatible to older versions, there are a few breaking changes that could impact older code. See here: https://blog.jquery.com/2016/06/09/jquery-3-0-final-released/

Tip! Rename the downloaded jQuery library from its full name (jquery-1.4.2.min.js) to just jquery.js so when you download updated versions you won't have to edit all of your site’s pages.

### Linking to the Library

Here's an example of a link to a jQuery library stored on your server:



or if you follow the renaming tip above:



This link can be used in each page as needed, or loaded once from a master or layouts page.

**Linking to an external library:**

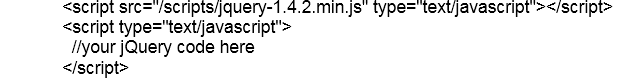
If you don't want a local copy of the library, you can link to a file stored in a common location on the internet. You can take advantage of Content Delivery Networks (CDNs) as public locations for commonly used files such as jQuery. Be aware that if you go this route, your users will always need access to the internet while accessing your site. While this may be a disadvantage for users on your local network, this can reduce load on your web servers when supporting thousands of internet users. But also consider that these remote sites are outside of your control and could be changed, compromised or deleted at any time. CDNs can be free, available for a fee or supplied by your hosting service. Here's a sample of loading the library from Microsoft's AJAX CDN site:



**Where should you put the link to the library?**

If you only need the library "here and there" then only load it in the pages that need it. If you will be using jQuery in many pages, then consider adding the library link to the site's master or layouts page. Whichever you choose, don't do both or you may get errors.

Add the library:



To add the library in the master page, just add the script link somewhere in the <HEAD> section of the page.

## “jQuery.” vs. “$.”

The "$" is a shortcut for "jQuery". Typing "jQuery" in place of the "$" will always work. The reverse is not always true. Historically, the “$” has been often used as a one character name for a JavaScript library or function. If you have an existing library that uses “$”, you can call $.noConflict() and jQuery will “return” the “$” back to the other library. You will then need to use “jQuery” in place of “$”. For more on the use of “$” and how to replace it with another name see: https://api.jquery.com/jQuery.noConflict

## Controlling When Your Code Runs

To make sure that all of a page’s objects have been added to the DOM before your JavaScript code attempts to interact with them, you could place all of your JavaScript code at the end of the page. While this is often done, you might prefer to load your JavaScript in the head section of the page, or you might want to load the JavaScript with a web part, web control or add-in that would place them in the middle of a page.

Browsers include a “window.load” event that fires when all page assets, including images, have been loaded. You could use this event to call your JavaScript code, but you would need to wait until all of the images have been downloaded before calling the code. jQuery includes a “.ready” method that fires as soon as the DOM is loaded that does not wait on images. If you need to know the dimensions of an image before running your code, then use “window.load”. If you need to update the DOM as quickly as possible then use the jQuery “.ready” method.

While you will find many examples of the “.ready” method using selectors on all kinds of objects, as of jQuery 3.0 only the following is recommended syntax:

* $().ready( *yourHandler* )

The following are examples you might find on the web. They are all equivalent, and are still supported:

* $( *yourHandler* )
* $( document ).ready( *yourHandler* )
* $( "document" ).ready( *yourHandler* )
* $( "img" ).ready( *yourHandler* )

The “*yourHandler*” above is either a function name or an anonymous function.



Or



For more on .ready() see: https://api.jquery.com/jQuery.ready/

## Selecting Elements using jQuery

jQuery is a query tool, and searching for and retrieving HTML objects is what jQuery is best at. When searching for tags based on CSS properties, jQuery uses the same notation as CSS to select elements: "." for classes, "#" for IDs etc.

Tags are selected by adding the tag name in quotes:



Tags with IDs are selected using "#":



Tags assigned to a CSS class are selected using ".":



Tags can be selected with combinations of selectors just like in CSS:



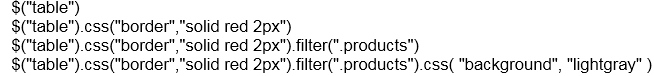
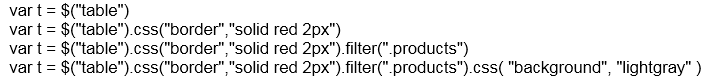
This last example selects all anchor tags inside of elements assigned the class named “sitenav”.

As of 3.2.1, jQuery has over 60 CSS-like selectors. See: https://api.jquery.com/category/selectors/

## Working with the Data Returned by jQuery

jQuery will return an array-like object that contains the matched HTML elements. You can store the results of a "query" in a variable or directly update all of the objects found in a single step by using jQuery methods.

jQuery returns the collection in two ways.

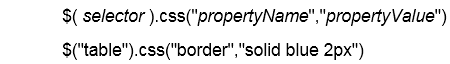
* The returned collection can be immediately accessed by jQuery functions by using dot notation. The first example returns a collection of tables while the second also returns a collection of tables that are now styled with red borders. The third example returns a smaller set of tables where the table has a class name of “products”.  
  
* The result of the final method in the jQuery statement is returned and can be stored in a variable.  
  
  + Note that if there are four tables on the page, two with a class name of “products”, the first two examples return four tables and the last two returns two tables. Study the last example… it retrieved four tables, set the border of those four tables, then filtered based on class name and the changed the background of two tables. I.e. One statement changed the borders of four elements and the backgrounds of only two. Also note that the variable “t” only contains the two tables with a class name of “products”.

Important! When performing a jQuery query ($(*selector*)), jQuery always returns a collection, even if it found only one or zero items. The jQuery methods deal with this and do not raise an error. For example, $("table").css("border","solid red 2px") will not raise an error if there are no tables or only one table. If you really need to return a single object, then use: var x = $("#*someid*").get(0); “get()” selects the nth item from the collection where 0 is the index of the first item.

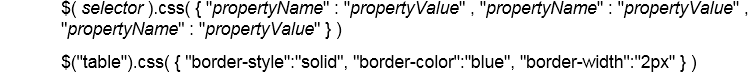
Important! (And should be obvious from the previous paragraph…) Some jQuery functions return collections of elements while some return values.

## Setting CSS Properties

To set a single property:



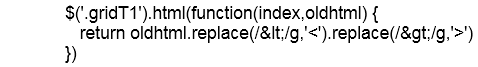
To set multiple properties (note the curly brackets and the colons):



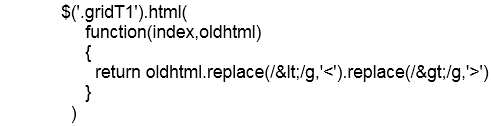
As of 3.2.1, jQuery has 18 CSS related methods. See: https://api.jquery.com/category/css/

## Running a Function Against the jQuery Return Set

jQuery does not have a built-in function for everything you can think of. For example, if you need to do a double search and replace, replacing both "<" and ">" in the example here, you might write an anonymous function and call it from one of the jQuery methods. This will call the function for each item found by jQuery.



or with a little better formatting:



If you do a search at jQuery.com for ".html" you will find that it can set the innerHTML from a string or call a function to set the innerHTML.



Here "index" is the position of the item in the return set and "oldhtml" is the original value of the HTML in the element in the returned item in the set.

### .each()

jQuery also includes a generic iterator that can call a function for each item in a collection. The keyword “this” is used to represent the current item in the collection.

Display a message for each table found:



Add a new row to every table:



The second example wraps “this” with “$()” so we can use jQuery methods such as “append”.

See: https://api.jquery.com/each/

## jQuery DOM Manipulation Methods

jQuery includes a number of methods to add and remove DOM elements.

* **Add elements around existing content:** .unwrap(), .wrap(), .wrapAll(), .wrapInner()  
   See: https://api.jquery.com/category/manipulation/dom-insertion-around/
* **Add elements inside of an existing element:** .append(), .appendTo(), .html(), .prepend(), .prependTo(), .text()   
   See: https://api.jquery.com/category/manipulation/dom-insertion-inside/
* **Add new content outside of an existing element:** .after(), .before(), .insertAfter(), .insertBefore()  
   See: https://api.jquery.com/category/manipulation/dom-insertion-outside/
* **Remove elements:** .detach(), .empty(), .remove(), unwrap()  
   See: https://api.jquery.com/category/manipulation/dom-removal/
* **Replace elements:** .replaceAll(), .replaceWith()  
   See: https://api.jquery.com/category/manipulation/dom-replacement/
* **Copy elements:** .clone()  
   See: https://api.jquery.com/category/manipulation/copying/

## jQuery Effects and Animations

jQuery includes a collection of methods to support fading, sliding, hiding, showing of elements along with a methods to perform CSS animations.

* **Basic:** .hide(), .show(), .toggle()  
   See: https://api.jquery.com/category/effects/basics/
* **Fading:** .fadeIn(), .fadeOut(), .fadeTo(), fadeToggle()  
   See: https://api.jquery.com/category/effects/fading/
* **Sliding:** .slideDown(), .slideToggle(), .slideUp()  
   See: https://api.jquery.com/category/effects/sliding/
* More: https://api.jquery.com/category/effects/custom-effects/

## jQuery Utility Methods

jQuery includes 37 (as of 3.2.1) utility or “helper” methods that might save you some code or produce more readable code. (See: <https://api.jquery.com/category/utilities/>)

A few of interest:

* **$.each()** – A generic iterator function that works on both arrays and jQuery collections. For each item in the collection .each() calls a function. (not the same as $(selector).each()) See: https://api.jquery.com/jQuery.each/
* **$.now()** – Equivalent to (new Date).getTime()
* **$.parseXML** – Parses a string of XML into an XML Document object without having to test browser versions. See: https://api.jquery.com/jQuery.parseXML/
* **$.trim()** – Removes spaces from both the beginning and end of a string. See: https://api.jquery.com/jQuery.trim/
* **$.type()** – Returns the internal JavaScript class of an object. See: https://api.jquery.com/jQuery.type/
* **Several array helpers:** .makeArray(), .map(), .merge(), isArray(), .inArray()

## jQuery Plug-ins

There are thousands of jQuery plug-ins that extend the functionality of jQuery. Once added to a page, plug-in code looks just normal jQuery.



The core jQuery team has created these plug-ins:

* **jQuery UI** – a library of “widgets” and classes to create accordions, fancy buttons, calendars, data pickers, progress bars, sliders, spinners and must more. See: https://jqueryui.com/
* **jQuery Mobile** – a framework to build mobile applications. See: https://jquerymobile.com/

There are repositories and lists of jQuery plug-ins on the web:

* http://plugins.jquery.com/ (no longer being updated)
* https://www.npmjs.com/browse/keyword/jquery-plugin

## Useless Trivia!

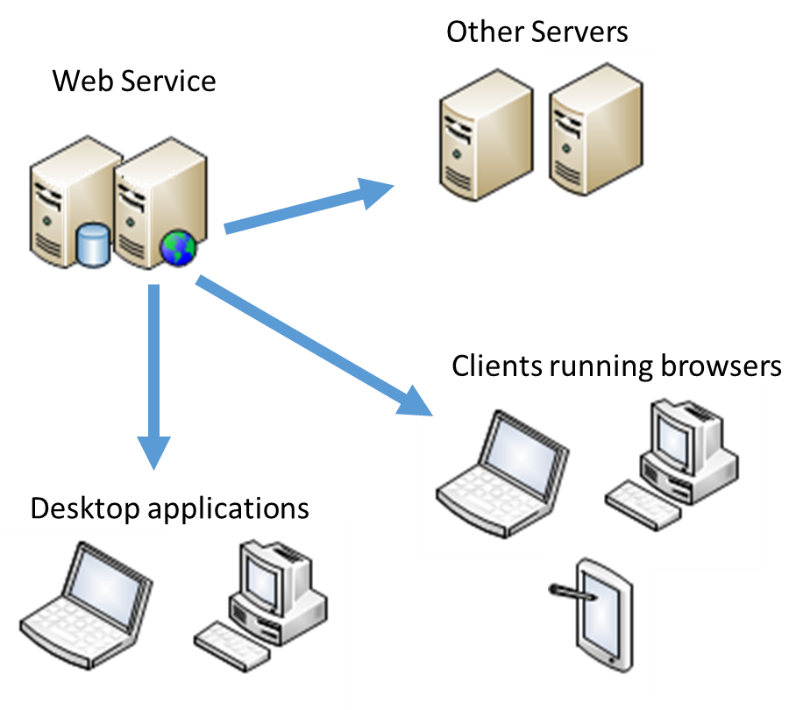
* jQuery was released in 2006.
* You can view jQuery as a work in progress at https://github.com/jquery/jquery.
* Microsoft includes jQuery with Visual Studio for ASP.NET AJAX and MVC project templates.
* According to builtwith.com, over 700,000 web sites are now using jQuery.
* jQuery (as of 3.2.1) has 140 built-in functions/methods. (Not counting any plug-ins.)

# Module 4: Ajax and Web Services

## Web Services

Web Services Definitions:

* If you think of a web page as an HTTP machine to human communication, then a web service is an HTTP machine to machine communication. The machines could be anything network connected and includes web browsers.
* While a web site typically returns web pages, a web service returns data. The data could be in text, JSON, XML or another format.



Characteristics of Web Service

* Server - Client based.
  + The server receives requests using HTTP with the query defined by the URL, an HTTP header or a payload (XML for example).
  + The server returns data to client in text format. This could be simple text, encoded binary data, XML or JSON.
* Can be hosted on local servers, cloud servers or local PCs.
* Stateless - No context is stored between requests.
* Can be private and secured, or public and anonymous.
* Often used as the "API" into applications such as SharePoint.
* Often cacheable.

**A Brief History of Web Services**

1. Remote Procedure Calls (RPC) - Exchanges of files (FTP, etc.) (way back when!)
2. Electronic Data Interchange (EDI) - Uses proprietary data formats (mid 1970's) and formalized using XML (mid 1990's).
3. HTTP and XML (SOAP) based web services (2000)
4. Representational State Transfer (REST) (First defined in 2000 and later by W3C in 2002)

## HTTP Verbs

In your web development work you may have used the HTTP verbs “GET” and “POST” and probably only associated those with URL Query Strings and form <form> submissions. These verbs are part of a larger set that define the basic Create, Read, Update and Delete set of data activities (CRUD).

* **POST** – this is the **C**reate activity, or the equivalent to a SQL Insert statement.
* **GET** – This the **R**ead activity, or the equivalent to a SQL Select statement.
* **PUT/PATCH/MERGE** – this is the **U**pdate activity, or the equivalent to a SQL Update statement.
  + In REST PUT is meant to replace the entire entity while PATCH/MERGE is used to update part of an entity.
* **DELETE** – this is the **D**elete activity, or the equivalent to a SQL Delete statement.

Notes:

* A GET action is easy to manually test as it is just a URL typed into a browser. The other actions will require writing code or by using a tool like Fiddler.
* More: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

## Data Formats

While a web service can return almost any kind of HTTP compatible data, the most common formats used to represent objects are XML and JSON. XML and JSON are intended to be both human readable and machine readable. Objects from JavaScript and other languages cannot be sent using HTTP as-is. The object needs to be serialized into a text format for transmission and then deserialized from text by the consumer of the web service.

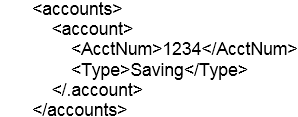
### XML

e**X**tensible **M**arkup **L**anguage

XML was first released in 1996 and the XML 1.0 standard was released in 1997. “1.0” is the latest version in common use today! (There is a 1.1, but it is not widely used.)

* XML is a tag based markup language with no pre-defined tags. <mytoys><toy id="1" name="boat"/></mytoys> is perfectly valid.
* XML dialects have been created for standardized objects. For examples see: https://en.wikipedia.org/wiki/List\_of\_XML\_markup\_languages
* XML is case sensitive.
* XML is used to define many .NET Framework files such as web.config.

Example:



The XML structure will depend on a schema, which might require attributes instead of elements, or a mixture of elements and attributes. The following uses XML attributes to represent properties.



### JSON

**J**ava**S**cript **O**bject **N**otation

While defined in the early 2000’s, JSON became an ECMA standard in 2013. JSON is similar to the syntax used to create a new JavaScript object.



JSON is a string format that follows that pattern.



The string is converted into a JavaScript object using the JavaScript JSON.parse() method.

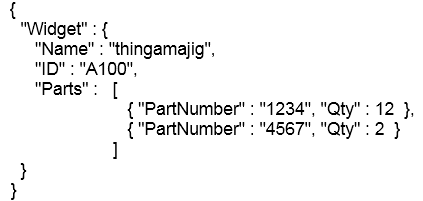


HTTP protocols send and receive text, or text encoded, data. JavaScript objects can be converted (serialized) to strings using the JSON.stringify() method.



The tools needed to parse JSON are built into JavaScript. Similar tools are available for other languages, for example the Newtonsoft.Json library for the .NET Framework.

The JSON Syntax:

* Properties are defined as name/value pairs separated with a colon.  
  
* JSON defines objects using curly brackets. Multiple properties are separated by commas.  
  
* Collections of objects are defined with square brackets.  
   
* Objects can have a hierarchal collection of properties. For example, Widget has three properties named Name, ID and Parts. Parts is a collection of part objects.  
  

JSON strings are typically code generated and are not usually “pretty formatted”. This is done to reduce the size of the data transferred by removing unneeded whitespace characters.  


Tools to reformat or validate JSON:

* Visual Studio has a built-in editor for files with a .JSON extension and supplies IntelliSense support inside of JavaScript code blocks.
* Bing search even has a JSON formatter:  
  https://www.bing.com/search?q=JSON%20formatter

#### JSON and Dates

JSON does not know what a date datatype is, and each JSON serializer will return its own favorite date format. Here are some examples:

* "\/Date(1489627020170)\/" is a JavaScript serial number for number of milliseconds since January 1, 1970.
* "\/Date(1489627020170-0500)\/" is the same, but with a time zone adjustment.
* "2017-03-15T17:46:02.1153043" is the ISO8601 standard with a time zone adjustment.
* "2017-03-15T17:46:02.1153043-07:00" is the ISO8601 standard with a time zone adjustment.

##### Dates and the JavaScript Client

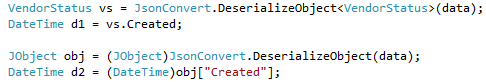
The first example, "\/Date(1489627020170)\/", is what is returned by the Microsoft MVC Controller’s Json() method. To do anything useful with this on the client side using JavaScript we will need something like this:  


The third example, "2017-03-15T17:46:02.1153043", is what is returned by Newtonsoft.Json. JavaScript’s JSON.parse() will still convert this to a string, but it’s a little easier to use if converted to a date type.



##### Dates and the C# Client

The Newtonsoft.Json library will properly map from a JSON date-like property and a .NET object’s DateTime property using either a cast to your type (VendorStatus in the example below) or to a JObject collection.



What if you are using a web service that returns "\/Date(1489627020170)\/" and your C# client code does not understand that format? You will have to take those milliseconds and add them to 1/1/1970!



If you need to create the JavaScript date number from C# you could do this:



**Other Clients**

Most languages and tools will either include native support for JSON, or can use third party libraries. You will need to explore how they handle JSON dates.

## Ajax

Many web services can be accessed from client-side code using JavaScript. Most of these calls will be asynchronous to prevent the user’s browser from freezing during updates. In JavaScript we use Ajax to make asynchronous calls to remote web servers.

**Synchronous vs. Asynchronous**

* A synchronous call is a “blocking call”. When the line of code is reached that performs the call, the code will pause until call has completed, typically when data has been returned. No other code can run until the call has completed. The JavaScript alert() function is an example of something that blocks.
* An asynchronous call does not block. As soon as the call is made the JavaScript continues with the next line of code. Asynchronous code usually includes a callback function that is called when the asynchronous code has completed and another that is called if an error has occurred.

Ajax used to be called AJAX (**A**synchronous **J**avaScript **A**nd **X**ML). While it uses JavaScript, it does not have to be asynchronous or return XML. Today Ajax is a name for a technology and is not usually in all upper case.

Ajax:

* Provides communication to and from web servers.
* Makes asynchronous calls so as to not block other code running in the browser.
* Can also make synchronous / blocking calls.
* Can send and receive text, HTML, XML, JSON and other data.

### Calling Web Services using JavaScript

Microsoft supplied an ActiveX object named XMLHTTP that let earlier versions of Internet Explorer send and receive data asynchronously. Other browser vendors followed up with a native XMLHttpRequest object. This object was finally standardized in HTML 5 as the window.XMLHttpRequest object.

An example:

Create the XMLHttpRequest object:



Create the asynchronous event handler to process the results of the call:



Specify how to make the call (GET, POST, PUT, DELETE, etc.) and the URL of the web service:



Make the call:



The send method can include data needed by the server to process the request. This could include data to create a new customer, credentials, etc.

You will usually add code to deal with timeouts and errors.

### Calling Web Services from Ajax and jQuery

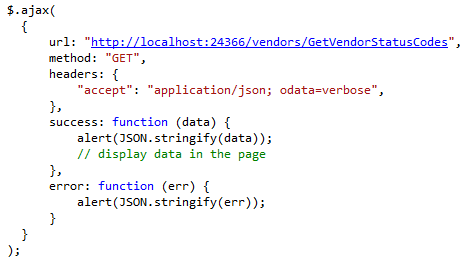
If we write our web service calls using the XMLHttpRequest object we will also need to write code to deal with multiple browsers and browser versions. While there are numerous Ajax libraries available, in this course we will use the jQuery library. jQuery lets you create a detailed request with many options, or a simple request to just get JSON data.

#### $.ajax()

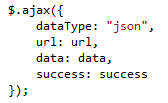
The Ajax method gives you the most control and the most options. See: <http://api.jquery.com/jQuery.ajax/>

In the example below:

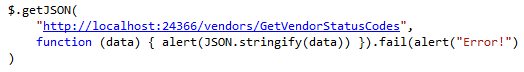
* The curly brackets define a JavaScript object that is passed to the $.ajax method.
* A custom HTTP header has been added to request JSON formatted data.
  + The service you are calling might be able to request multiple formats.
  + This will not guarantee that you will get JSON back. That’s up to the service.
* Asynchronous functions for processing a success or an error. This example is using anonymous in-line functions. (An alert is not the best way to show data!)
* The success property is the asynchronous callback function for the returned data. It receives a parameter that represents the returned data. The returned data could be in text, JSON or XML formats, depending on the web service.
* The error property is the function called when there’s an error. It receives and error object with details about the error.



#### $.getJSON()

The getJSON method is just a shorthand for:  
 

The following is the equivalent to the $.ajax example above.



$.getJSON has two shortcomings:

* If the JSON is malformed, it will “silently fail”.
* You do not get the cause of the error from the .fail() method.

The solution to these issues is to use the full $.ajax() method.

## RESTful Web Services

**RE**presentational **S**tate **T**ransfer

Traditional web services do one thing and only one thing. Process a single unique request. They may accept one or more parameters to filter content, but they cannot be used in ways other than what they were written for. A GetVendors service might return all active vendors. If someone needs a list of all vendors from Ohio, or all vendors with for a particular product, then we need to write yet another web service. And as soon as that is delivered, they will want one that returns vendors for a ZIP code or a phone number prefix.

A REST service URL points to a resource, not a page, file or service.

* ASP.NET ASPX SOAP web service:  
   http://yourserver/SomeService**.asmx** a page, not a resource – returns a vendor or a product or a …?
* Windows Communications Foundation (WCF):   
   http://yourserver/SomeService**.svc** a page, not a resource – returns a vendor or a product or a …?
* REST  
   http://yourserver/Vendor no file extension! Clearly a request for a vendor!  
  Note: Web API routes will typically include a path prefix:  
   http://yourserver/**api/**Vendor

While REST services with OData can support full CRUD activities, its most useful feature is the ability to query data from a URL.

* http://yourserver/VendorsData?$select=name, phone&$filter=state eq 'OH'
* http://yourserver/VendorsData(1)/Products?$select=description, price

### REST:

* Hides the implementation
  + No .asmx or .svc files or pages in the URL (ideally)  
     http://site/resource/action?parameters
* A RESTful service models data entities.
  + Entities like Products, Customers, Announcements, Tasks, etc., not a method such as GetCustomer(). (Entities could equate to a table in SQL, but could refer to a customer and their orders.)
* Uses standard HTTP verbs for CRUD.
  + POST, GET, PUT, DELETE plus PATCH/MERGE
* Readable, predictable URLs.
* Typically returns JSON or XML.

## OData

OData is a standard that defines both a query syntax and a format for the returned data. OData can also generate its own documentation data.

For more about the OData standard see:

* **OData in ASP.NET Web API**  
  https://docs.microsoft.com/en-us/aspnet/web-api/overview/odata-support-in-aspnet-web-api/
* **Understand OData in 6 steps**  
  http://www.odata.org/getting-started/understand-odata-in-6-steps/
* **OData - the Best Way to REST**  
  http://www.odata.org/
* **Open Data Protocol** (with some history)  
  https://en.wikipedia.org/wiki/Open\_Data\_Protocol

Note: ASP.NET’s Web API implementation of OData does not fully comply with what’s described at the odata.org site.

### Service Metadata Document

OData can be self-documenting and can generate an XML document that defines the list of entities supported.

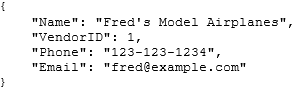
http://yourserver/odata**/$metadata**



### Data Format

While Web API by default just returns an XML or JSON representation of an object, OData returns that data plus “metadata” about the object.

**A Web API request for a single vendor:**http://yourserver/api/vendorsdata/1

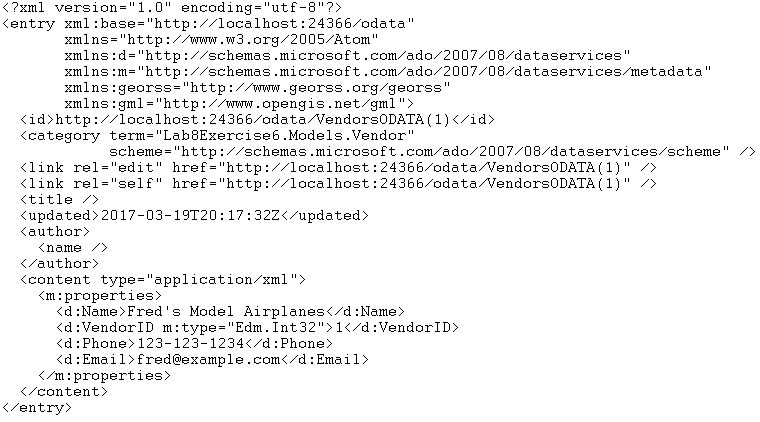


Web API by default returns just the data, and your object is the top level object. If you stored this into a JavaScript variable named yourVar you would access Name as yourVar.Name.

A REST request with OData formatted return for a single vendor can be returned in several formats  
http://localhost:24366/odata/VendorsODATA(1)

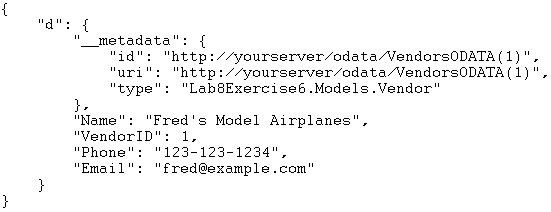
**OData Atom Pub (XML):**





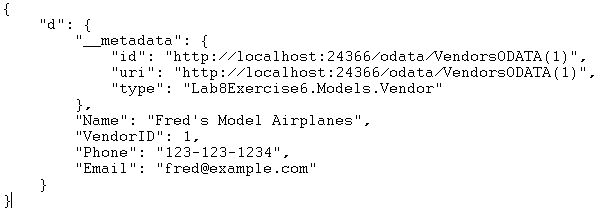
**OData JSON Verbose:**





**OData JSON Light (the default starting with ODATA 3)**





OData by default returns your data and metadata about the data. Your object is a property of a top level object named “d”. If you stored this into a JavaScript variable named yourVar you would access Name as yourVar**.d.**Name.

### OData is very case sensitive

All OData URLs and queries are case sensitive past the server name and any “/odata/” prefix.

* These are all valid:
  + http://yourserver/odata/VendorSerach?$select=Name
  + http://YourServer/oData/VendorSerach?$select=Name
  + http://YOURSERVER/ODATA/VendorSerach?$select=Name
  + http://YourServer/odata/VendorSerach?$select=Name
* Only one of these is valid:
  + http://yourserver/odata/VendorSerach?$select=Name
  + http://yourserver/odata/vendorserach?$select=Name
  + http://yourserver/odata/VendorSerach?$select=name
  + http://yourserver/odata/VendorSerach?$Select=Name
* Which of the above is the correct one? The fourth is never correct. “select” and other keywords are always in lower case. For the first three it depends on how things were named in the controller.

## OData Queries

OData URLs are first RESTful, pointing a resource, and then queryable by using a query string.

URLs that point to a resource:

* http://yourserver/odata/VendorsData (returns all vendors – a collection of resources)
* http://yourserver/odata/VendorsData(1) (returns a single vendor)
* http://localhost:24366/odata/VendorsODATA(1)/Products (returns a single vendor’s products)

URLs with a query:

* http://yourserver/odata/VendorsData?$select=Name (returns all vendors names)
* http://yourserver/odata/VendorsData(1)?$select=Name (returns a single vendor’s name)

#### Passing an ID

OData URLs pass parameters a little differently. While a Web API URL would select a vendor with ID=3 using a route, OData passes the parameter as you would for a method call.

* Web API: /api/Vendors/3
* OData: /odata/Vendors(3)

#### Queries

OData queries are written as a URL query string and follow a pattern:

* Starts with a “?”, if the first item in the query string, or with a “&”. ?$select=…
* Keywords start with a “$”. $select=…
* Keywords are followed by an equal sign.
* Boolean operators are names, not symbols. eq, ne, gt, lt, etc.

#### $select

$select selects which properties to include in the response.

Notes:

* Property names can be separated by spaces. $select=Name,Status or $select=Name, Status
* Some ODATA providers may still return additional properties beyond those in your list.

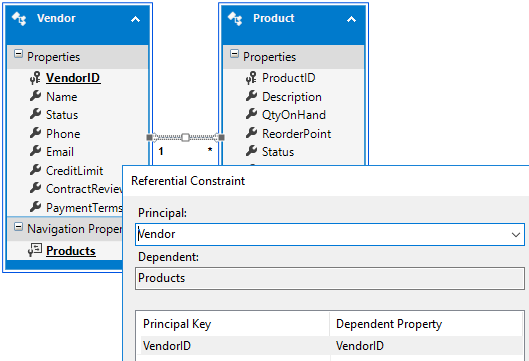
Examples:

* /VendorsODATA?**$select**=Name, Status
* /VendorsODATA(1)?**$select**=Name, Status

#### $select with drilldown

$select combined with $expand can be used to return parent/child sets of related data.

For this set of entities:



You could return a list of products for a vendor, or all vendors, using:

* http://yourserver/odata/VendorsODATA(1)?**$expand=Products**&$select=Name, Status, **Products/Description, Products/Price**
* http://yourserver/odata/VendorsODATA?**$expand=Products**&$select=Name, Status, **Products/Description, Products/Price**

#### $filter

$filter Filters the results, based on a Boolean condition.

For details and examples see: http://www.odata.org/documentation/odata-version-3-0/odata-version-3-0-core-protocol/#queryingcollections

Examples:

* http://yourServer/odata/VendorsODATA?**$filter=Status eq 'Active'**
* http://yourServer/odata/VendorsODATA?**$filter=substringof('plane',Name)**
  + This is the equivalent of a “contains” function.
  + The searched text (“plane”) is not case sensitive.
* http://yourServer/odata/VendorsODATA?$**filter=year(ContractReview) eq 2018**

Notes:

* $filter can only be applied to collections, not single objects.
* All keywords are case sensitive!
* All literal text must be wrapped in single quotes.
* Boolean operators are names, not symbols: eq, ne, gt, ge, lt, le, and, or, not
* String functions: length, substringof, startswith, endswith, indexof, replace, substring, tolower, toupper, trim, concat
  + $filter=length(Name) gt 25
  + $filter=startswith(Name,'fred')
  + Note: Text used in these functions is not case sensitive.
* Arithmetic Operators: add, sub, mul, div, mod
  + Example: $filter=Price sub 10000 gt 0
  + Example: $filter=VendorID mod 2 eq 0 only returns even numbered vendor IDs
* Date functions:
  + year(*date*)
  + month(*date*)
  + day(*date*)
  + hour(*date*)
  + minute(*date*)
  + second(*date*)
* Some of the OData implementations do not support aggregates such as sum and average.

#### $orderby

$orderby sorts the results. Uses a list of properties and can include “desc” or “asc”.

Example:

* $orderby=Name
* $orderby=CreditLimit desc,Name

Notes:

* $orderby can only be applied to collections, not single objects.

#### $top

$top Returns only the first n the results.

Example:

* $top=20

Notes:

* $top can only be applied to collections, not single objects.

#### $skip

$skip Skips the first n results. Typically used with $top and $inlinecount for paging of data.

Example:

* $skip=80&$top=20

Notes:

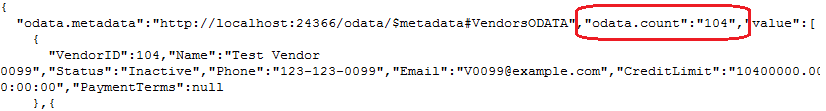
* $skip can only be applied to collections, not single objects.

#### $inlinecount

$inlinecount Tells the server to include the total count of matching entities in the response. (Useful for paging.)

Example:

* $inlinecount=allpages



Notes:

* The returned property is “odata.count”.
* $inlinecount can only be applied to collections, not single objects.
* $inlinecount has only two options: none and allpages.

## OData Updates

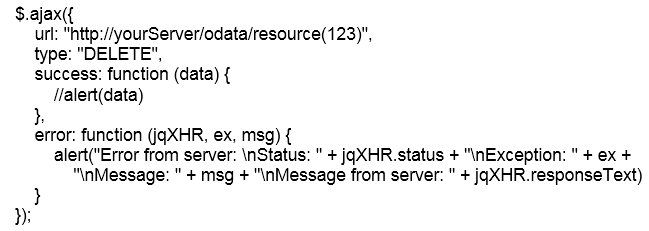
OData can update data using the HTTP POST, PATCH and DELETE verbs.

|  |  |  |
| --- | --- | --- |
| Verb | URL | HTTP body |
| DELETE | To resource: http://yourServer/odata/resource(id) | None |
| POST – insert | To resource: http://yourServer/odata/resource | Data needed to create a new item. Typically required fields.  JSON example: |
| PATCH/MERGE – update | To resource: http://yourServer/odata/resource(id) | Data to change.  JSON example: |

### Examples

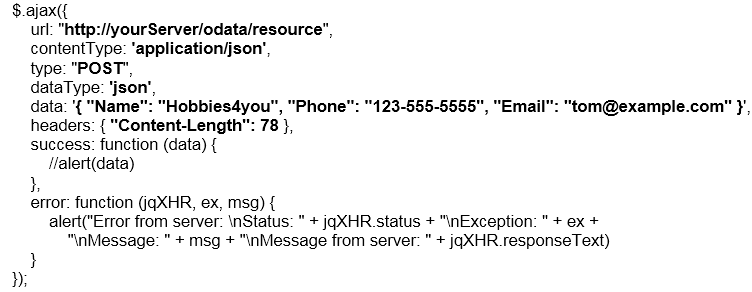
#### DELETE

An example using jQuery’s AJAX function to delete an object by ID.



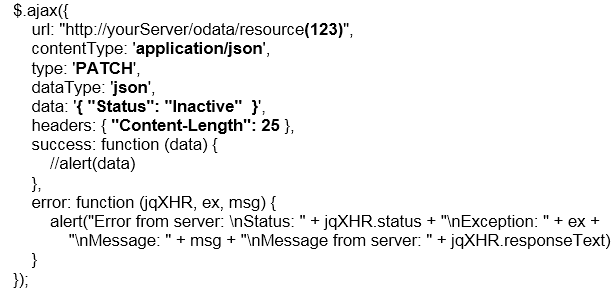
#### POST - Insert

An example using jQuery’s AJAX function to insert a complete object.



#### PATCH – Update

An example using jQuery’s AJAX function to set a “Status” property to “Inactive”.



# Module 5: Diving in Deeper!

In this module, we will explore the topics you will need to know as you move from a “JavaScript scripter” to a “JavaScript developer”.

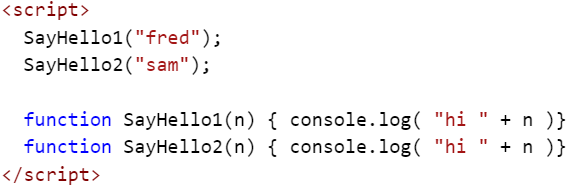
* More about Data Types and data conversion.
* Variables: more on Scopes, Hoisting, Creating and Deleting.
* Functions: Anonymous functions, Closures, Self-invoking functions, the “this” keyword and the Module Pattern.
* More on error handling.
* Recursion.

## Is JavaScript Compiled?

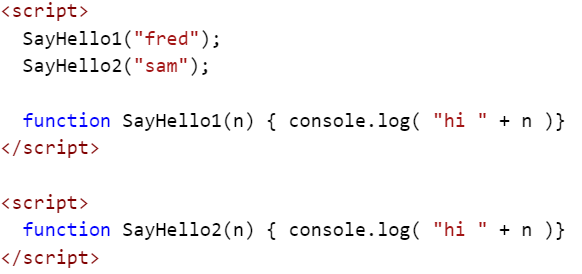
JavaScript code is reparsed each time the page is loaded. No binaries are created or saved anywhere. (In contrast, .NET compiles to .DLL files.) So, is it interpreted or compiled? Early versions of JavaScript were pure interpreters that parsed and executed the JavaScript code as it was read. Modern browsers optimize JavaScript by “just in time” compiling script blocks and then executing the compiled version.

#### Do I need to define a function before calling it?

When a script is parsed, it is compiled “just in time” (JIT) and then executed. In the example below, both functions will be properly called, even though they are defined after where they are called.

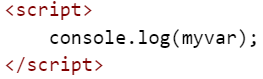


But, if you have two scripts (files or script blocks) they are compiled and executed one at a time. In the next example, the call to SayHello2 will fail as the second script has not been compiled before the first script was run.

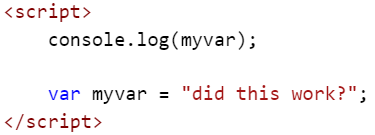


What about variables?

This gets interesting… The following code generates an error: “’myvar’ is undefined”. This is expected as it is not defined anywhere.

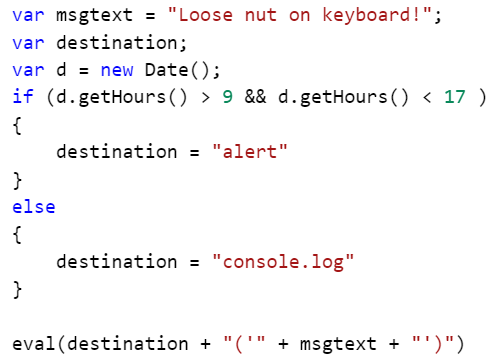


The next code sample does not generate an error! The message “undefined” was written to the console. The “compile pass” found the variable, but the “execute pass” has not yet found the variable initialized with a value. When “console.log” was executed, the variable “myvar” existed, but did not have a value.

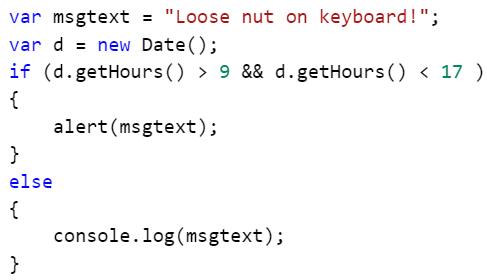


#### eval()

One aspect of JavaScript that is pure interpreter is the eval() function. Eval takes a string, parses it and then executes it.



Avoid eval() where ever possible. It’s too easy to create unexpected code and in some cases opens up your code to hackers. The above example can be rewritten into a much better form.



Notes:

* The .compile() method does not compile code! It compiles regular expressions. It was deprecated in JavaScript version 1.5.
* Google has a tool named “Closure Compiler” that rewrites and compacts JavaScript code to reduce download size. It does not compile code!
* There are a number of languages that compile into JavaScript.
  + List of languages that compile to JS  
    <https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS>

## Data Types and Objects

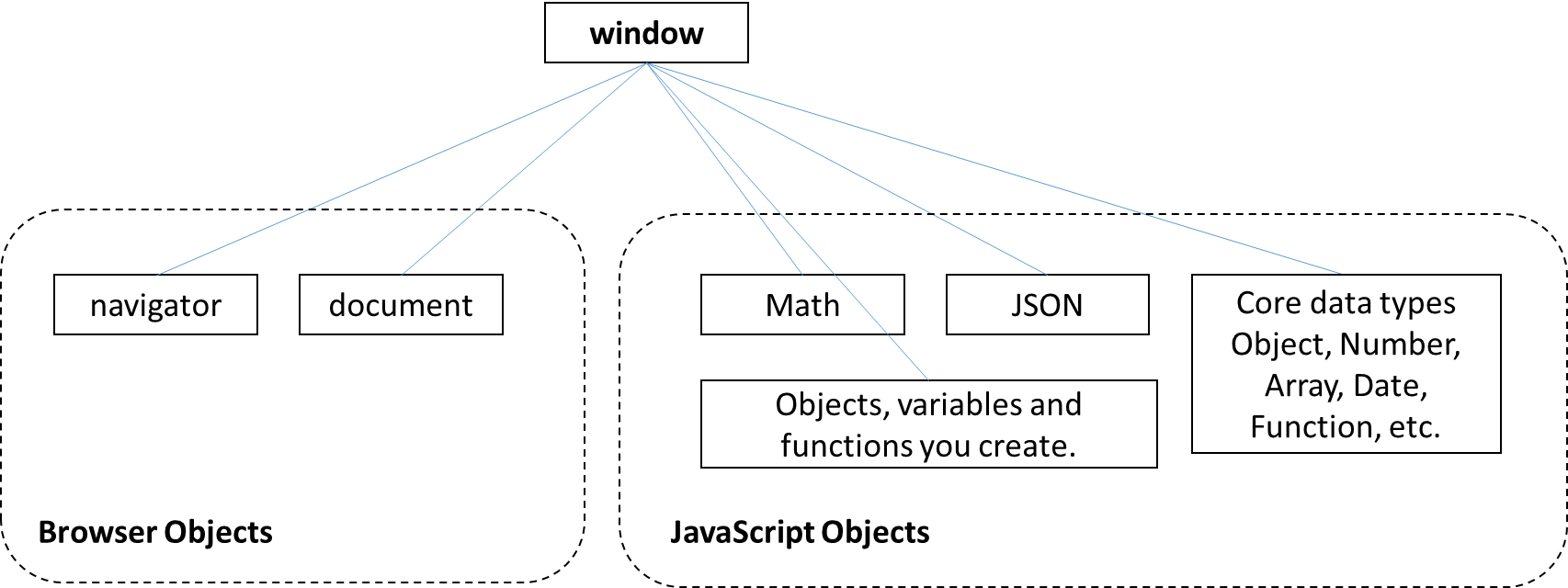
### JavaScript Primitives

A “primitive” is a simple type that only has a value.

* Values: (I.e. you cannot have a variable of type null! You can have a variable of another type that could be equal to null.)
  + null
  + undefined
* Types: (I.e. you can have a variable of type boolean.)
  + boolean
  + number
  + string

### JavaScript Default Objects

When working with web pages, JavaScript runs in the context of a browser. The top most object in JavaScript is window. The window object has properties that contain all of the other objects that JavaScript has access to.



* window – The top most object. Contains all other objects.
* window.navigator – The web browser and its configuration.
* window.document – The currently loaded document. This object is typically defined using an HTML file. The file is parsed to create the Document Object Model (DOM) that is accessible by JavaScript through the document object.
* window.Math – JavaScript’s math library (random, sin, cos, sqrt, etc.)
* window.JSON – JavaScript’s JSON / JavaScript Object Notation library. Used to serialize JavaScript objects (JSON.stringify()) and to deserialize a JSON string into a JavaScript object (JSON.parse()).
* window.Object, window.Array, window.Number, etc. – the core data types.
* window.*yourObjects* – all other objects created by your JavaScript code.

As window is the default object, you do not need to type “window” in front of any of the above objects.

Samples of properties for the window object:

* navigator.userAgent // "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:51.0) Gecko/…"
* navigator.userLanguage // "en-US"
* Math.sqrt(2) // 1.4142135623730951
* JSON.stringify(bankaccount) // {"AcctNum":1234,"Type":"Saving","Closed":false," … }
* document.doctype // <!DOCTYPE html>
* document.head.title // "Your site title"  
  document.title // "Your site title"
* document.getElementsByTagName("h1") // *collection of all H1 elements*

### Window Methods

The window object contains all of the built-in JavaScript methods. In the list below, each method could be written with the window object and a dot. Example: window.alert("hello world");

A partial list of frequently used window methods.

* alert("*message*") – displays a popup message.
* confirm("*message*") – displays a popup message with OK and Cancel buttons. Clicking OK returns true and clicking Cancel returns false.
* prompt("*message*","defaultValue") – Prompts the user for a string value. Clicking OK returns the string and clicking Cancel or pressing Esc returns null.
* console – returns the console object that displays messages to the browser’s console (F12 tools).
  + console.log("*message*") – writes a message to the console.
  + console.warn("*message*") – writes a warning message to the console.  
     console.warn("here be dragons") 
  + console.group() – indents all following console output. (can be nested)
  + console.groupEnd() – unindents all following output.
  + console.time("*timername*") – creates a timer.
  + console.timeEnd("*timername*") – displays the elapsed time and removes the timer.
  + console.trace() – displays the call stack to this point in time.
  + See: https://developer.mozilla.org/en-US/docs/Web/API/Console
* print() – opens the print dialog box for the current window.
* Also see: https://developer.mozilla.org/en-US/docs/Web/API/Window

### Navigator Properties

The navigator object supplies information about the browser, and services supplied by the browser such as geolocation. Many of these properties are “non-standard” and are inconsistently implemented across browsers. As an example, Internet Export, Firefox and Chrome all return “Netscape” for the navigator.appName property and they all return “Gecko” for navigator.product property.

* navigator.userAgent – The user agent string sent to the web server to identify the browser. (These are often untrustworthy. navigator.userAgent is being removed from the standards.) Here are a few examples:
  + **IE 11:** "Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; Zune 4.7; MS-RTC LM 8; .NET4.0E; InfoPath.3; rv:11.0) like Gecko"
  + **Firefox:** "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0"
  + **Chrome:** "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36"
  + **Notes:**
    - Each of the above claim to be Mozilla/5.0!
    - Each gives hints as to their real id: “Trident” is the IE HTML engine and “rv:11.0” is the IE browser version. IE says “like Gecko” while Firefox says “Gecko”.
    - Some specialized browsers and some search engines “lie”!
    - So… don’t try to identify the browser or version. Test for features instead!
* navigator.oscpu – Returns the version of the operating system. (Not supported by IE 11 or Chrome.)
* navigator.onLine – Returns true if the browser has access to a network.
* navigator.language – Returns the language set in the browser’s user settings. Example: “en-US” for US English.
* navigator.languages – Returns an array with all of the user’s preferred languages. navigator.language is the first element of the array.
* See: https://developer.mozilla.org/en-US/docs/Web/API/Navigator

### JavaScript Objects

JavaScript objects have properties. Each property has a name and can contain a value (a primitive), a function, or another object.

Properties can be defined:

* When the object is created: var account = { AcctNum: 1234, Type:"Saving" };
* At any time: account.Balance = 1234.5;

Properties can be:

* A primitive value. (number, Boolean, etc.)
* An object
* A function

Objects can be created as an “object literal”:

* An empty object:  
   var obj = {};
* An object with primitive properties:  
   var bankaccount = { AcctNum: 1234, Type: "Saving", Closed: false, Balance: 1234.5 };
* An object with both primitive and object properties:  
   var customer = { CustNum: 1000, Account: bankaccount, Manager: "Jones" };
* Note: The curly bracket notation used here is similar to the JSON format. JSON strings are often returned by web services and can be quickly converted to JavaScript objects using JSON.parse().
  + var jsonstring = '{ AcctNum: 1234, Type: "Saving", Closed: false, Balance: 1234.5 }';  
    var bankaccount = JSON.parse(jsonstring);

Objects can be created using the “new” keyword:

* new Object();
  + var bankaccount = new Object();  
    bankaccount.AcctNum = 1234;  
    bankaccount.Type = "Saving";  
    bankaccount.Closed = false;  
    bankaccount.Balance = 1234.5;

The “new” keyword can be used to create objects based on primitives and other objects, but this is rarely used.

* new Array();
* new Boolean();
* new Date();
* new Function();
* new Number();
* new RegExp();
* new String();

### Accessing Object Properties

JavaScript supports two ways of accessing an object’s properties, dot notation and square brackets. Due to the square brackets, an object can also be treated as a container of name/value pairs and can even be enumerated in a for loop.

* var bankaccount = { AcctNum: 1234, Type: "Saving", Closed: false, Balance: 1234.5 };  
  var customer = { CustNum: 1000, Account: bankaccount, Manager: "Jones" };
  + customer.CustNum // 1000
  + customer["CustNum"] // 1000
  + customer.bankaccount.Balance // 1234.5
* Retrieving all of an object’s properties: (Excluding prototype properties.)
  + var props = Object.getOwnPropertyNames(customer);   
    for (var v in props) { console.log(props[v]) }
* The existence of a property can be tested. Example where “ABC” was never defined:
  + bankaccount.ABC == undefined // true
* Note: If bankaccount.ABC is set to null then:
  + bankaccount.ABC == undefined // true
  + bankaccount.ABC === undefined // false
* New properties are added dynamically:
  + bankaccount.lastCheckNumber === undefined // true  
    bankaccount.lastCheckNumber = 123;  
    bankaccount.lastCheckNumber === undefined // false

### JavaScript Numbers

JavaScript only has one numeric data type, “number”.

Number FAQ:

* All numbers are 64 bit floating point numbers.
* Floating point numbers “***approximate***” decimal values.
  + .333\*100 = 33.300000000000004
  + 1/3 = 0.3333333333333333, but 10/3 = 3.3333333333333335
  + 3.3333333333333333 \* 3 = 10
  + 3.3333333333333335 \* 3 = 10
  + 3.3333333333333329 \* 3 = 10
  + 3.3333333333333339 \* 3 = 10.000000000000001
  + n = Math.sqrt(2) // 1.4142135623730951  
    n \* n = 2.0000000000000004
  + Math.sqrt(2) \* Math.sqrt(2) == 2 // false
* Maximum value: 1.7976931348623157e+308 Minimum value: 5e-324
  + These two values can be retrieved using Number.MAX\_VALUE and Number.MIN\_VALUE.
* 15 places of precession. Although JavaScript can represent very large and very small numbers, it can only do so to 15 places of accuracy
  + 15 digits: 999999999999999 = 999999999999999  
    i.e. 999,999,999,999,999 = 999,999,999,999,999 (don’t type “,” in JS numbers!)
  + 16 digits: 9999999999999999 rounded to 10000000000000000  
    i.e. 9,999,999,999,999,999 rounded to 10,000,000,000,000,000
  + 18 digits: 100000000000000001 gets rounded to 100000000000000000  
    i.e. 100,000,000,000,000,001 gets rounded to 100,000,000,000,000,000
* Up to 17 digits past the decimal point.
* The Math.round() function is useful for floating point number adjustments.
  + Math.round() rounds to the nearest whole number:
    - Math.round(3.49999) // 3
    - Math.round(3.5) // 4
  + To round to two decimal places, multiply the value by 100, round and then divide by 100.
    - Math.round(3.4444 \* 100) / 100 // 3.44
    - Math.round(3.6666 \* 100) / 100 // 3.67
* The .toFixed(n) method is useful when the result needs to be a **string** and have a fixed number of decimal places.
  + Numbers reformatted to strings with specified decimals:  
    
  + The string can be returned to a number using parseFloat() or parseInt().  
    

#### Infinity

JavaScript numbers also define Infinity and -Infinity. Numbers that exceed Number.MAX\_VALUE and Number.MIN\_VALUE return Infinity.

* Division by zero does not raise an error in JavaScript, it returns “Infinity”.
* Infinity is a number!
  + typeof Infinity; // “number”
* Infinity +, -, \*, / any number is also Infinity
  + Infinity \* 2 // Infinity

#### NaN

Invalid numeric operations return a special type called “Not a Number”, or NaN.

* NaN never equals anything!
  + NaN == 5 // false
  + NaN == NaN // false
  + NaN == null // false
* NaN is a number!
  + typeof NaN; // “number”
* Examples that return NaN:
  + 5 \* “a” // NaN
  + Math.sqrt(-1) // NaN

#### Number variables have methods

Everything in JavaScript is an object, and all objects have properties and/or methods.

* toExponential() – returns a string using exponential notation.
  + (12345.12).toExponential() // "1.234512e+4"
* toFixed(*places*) – returns a number as a string formatted to a fixed number of decimal places.
  + (12345.12345).toFixed(2) // "12345.12"
* toLocaleString(*locale*) – returns a number as a string formatted for a locale.
  + (12345.12).toLocaleString("fr") // "12 345,12"
  + (12345.12).toLocaleString("us") // "12,345.12"
  + (12345.12).toLocaleString("de") // "12.345,12"
* toLocaleString(*locale,options*) – returns a number as a string formatted for a locale.
  + (12345.12).toLocaleString("us", { style: 'currency', currency:'USD' } ) // "$12,345.12"
  + (12345.12).toLocaleString("de", { style: 'currency', currency:'EUR' } ) // "12.345,12 €"
* toPrecision() – Returns a string to the number of digits of precision.
  + var x = 5.123456; x.toPrecision(1); // "5"
  + var x = 5.123456; x.toPrecision(3); // "5.12"
  + var x = 5123456; x.toPrecision(3); // "5.12e+6"
  + var x = 5; x.toPrecision(3) // "5.00"
* toString() – returns a string representing the number.
* valueOf – returns the value of the variable.
  + var n = 5;  
    n.valueOf(); // 5

#### To and From HEX

* Numbers can be written in hexadecimal format by prefixing the value with “0x”:
  + var num = 0x10 // 16
  + var num = eval(“0x10”) // 16
  + var num = parseInt(“0x10”,16) // 16
  + Tip: parseInt(*str*,*base*) can be used to convert a string to a number.
    - var num = parseInt(“1100”,2) // 12
    - var num = parseInt(“14”,8) // 12
* Numbers can be converted to a hexadecimal string using .toString():
  + var hexstring = num.toString(16) // “0x10”
  + Tip: .toString(*n*) can convert a number to any base! (“n” must be 2 to 36)
    - n=12;
    - n.toString(2) // 1100
    - n.toString(8) // 14
    - n.toString(10) // 12
    - n.toString(16) // c

#### The Math object

JavaScript’s Math object includes a list of constants and math functions.

Constants: E, LN10, LN2, LOG10E, LOG2E, PI, SQRT1\_2, SQRT2

Functions: abs, acos, asin, atan, atan2, ceil, cos, exp, floor, log, max, pow, random, round, sin, sqrt, tan, min

See:

* https://www.w3schools.com/js/js\_math.asp
* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Math
* https://www.ecma-international.org/ecma-262/6.0/#sec-math-object

### JavaScript Strings

#### Escape Characters

You can embed special characters into strings using a backslash followed by a character.

* \0 Null
* \' Single quote 'It\'s raining!'
* \" Double quote "He said \"It's raining!\""
* \\ Backslash "Enter Unicode characters as \\uxxxx.”
* \n New line "this is on the first line \nthis is on the second line"
* \r Carriage return "this is overwritten by \rthis text" (may behave the same as \n.)
* \v Vertical tab Generally only applies to printers.
* \t Tab
* \b Backspace
* \f Form feed Generally only applies to printers.
* \uxxxx Unicode character "\u0061" = “a”
* Backslash followed by a carriage return lets you break long lines of text. The following creates a single long string:
  + var text = " Lorem ipsum dolor sit amet, consectetuer adipiscing elit. \  
    Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar \  
    ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna.";
  + Note: There must not be any other characters between the backslash and the carriage return.

#### String Properties

* .length – Number of characters in a string. (Not the number of bytes! JavaScript typically stores each character as two bytes.)

#### String Methods

Here are a few of the string methods:

* charAt(*i*) Returns the character at a specified index or position.
* includes(*str*) Returns true or false if *str* is found in the string.
* endsWith(*str*) Returns true of false.  
   (ECMAScript 2015, not supported by all browsers.)
* indexOf(*str*) Returns the index of the first occurrence of the string, or -1 if not found.
* lastIndexOf(*str*) Returns the index of the last occurrence of the string, or -1 if not found.
* match(*regEx*) Regular Expression match.
* repeat(*i*) Repeats a character or string.   
   (ECMAScript 2015, not supported by all browsers.)
* replace(*regEx|str, str*) Performs a searh and replace.
* search(*regEx*) Regular expression match.
* slice(beginIndex,endIndex) Extracts a substring by start and end positions.
* split(*str*) Splits a string into an array.  
   "red, gren, blue".split(",") = ["red","green","blue"]
* startsWith(*str,i*) Returns true if the string starts with *str*. “*i*” is an optional start position.  
   (ECMAScript 2015, not supported by all browsers.)
* substr(*i, cnt*) Returns the string starting at *i* and including *cnt* characters.
* substring(*i, i*) Returns the string between to indexes.
* toLowerCase() Returns a lowercase version of the string.
* toUpperCase() Returns an uppercase version of the string.
* trim() Returns a string with white space (spaces, tabs, etc.) removed from both ends. (Not supported by all browsers. Use a regular expression or a polyfill. There is a polyfill here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/String/Trim)

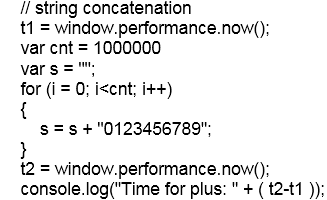
For more string methods see:   
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/String

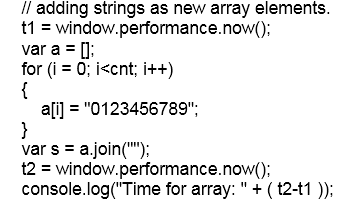
Notes:

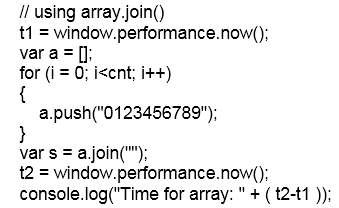
* Strings are indexed and start with 0. A string with a length of 10 has character 0 through character 9.
  + var s = "abcdefghijk";
  + s[2] // “c”
* .slice() can use negative numbers to count from the end of a string.
* Strings are immutable. They cannot be changed, rather they are copied with any modification and the original then disposed.

#### String Concatenation vs. an Array of Strings

In some cases you can speed up string concatenation by using an array of strings, followed by a .join(). Running the sample below in your collection of browsers may produce some surprising results. In tests made at the time this writing, an array is faster in IE while string concatenation is faster in Firebox. Regardless of the browser, an array.push() was about the same speed as string concatenation.



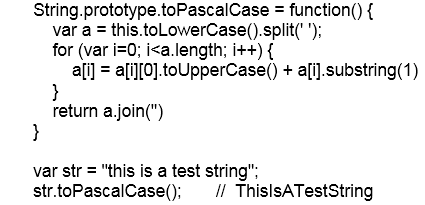




## Extending Data Types

Each object includes a “prototype” property that defines each of the objects properties. You can extend existing data types by adding a new prototype.

As an example, strings have toUpperCase and toLowerCase methods, but not a toPascalCase method. You can add your own toPascalCase!

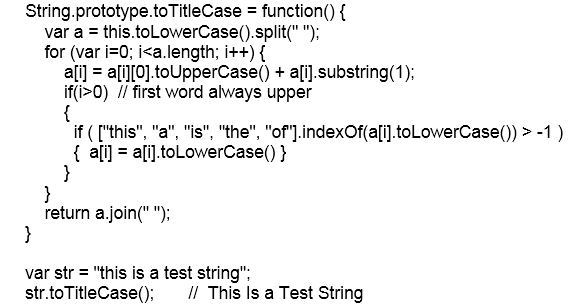


Notes:

* We are extending the base “String” object, not a single instance of a string. All strings in the page will have access to the function.
* The new method is available on all strings in the page.
* “this” is the current value of the string instance.

**More examples:**

Convert a string to “Title Case”. (Capitalize the first letter of each word, except for “noise words” like “a” or “the”.)



Double a number.

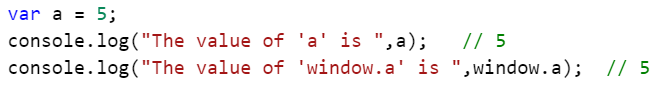


Round to nearest even number.



## Variables

In most languages, a variable is a pointer to a location in memory. In JavaScript, a variable is a property of some object. Global variables are properties of the window object.



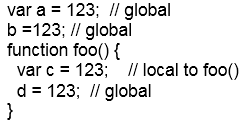
### Scopes

In JavaScript, variables have up to three levels of scopes.

* Global
* Function
* Block {}

#### Global Scope

Variables are added to the global scope when they are:

* Defined outside of any function with or without “var”.
* Defined within a function without using “var”.
  + Example:  
     
  + Variables created in functions without “var” are said to be “hoisted” to the global scope. These variables are properties of the window object.
  + If you add “use strict” to a scope, then attempts to create variables without declaring them with “var” will raise an error.

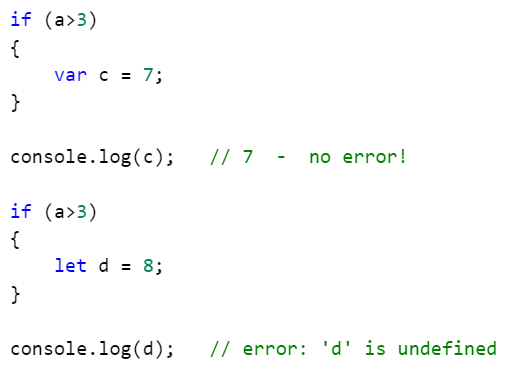
#### Function Scope

Variables are added a function scope when:

* Declared within the function’s code block ({}) using “var”.
* When not declared using “var”, the variable is created at the Global level.

#### Block {} Scope

ECMAScript 2015 (6) adds a “let” keyword that can be used in place of “var” to create variables that are scoped to a code block ({}) such as an “if” or “for”.

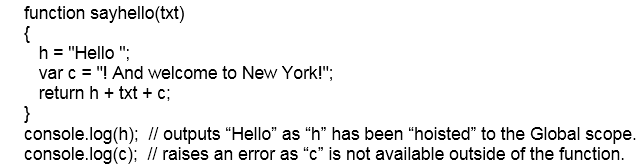


The “let” keyword is supported in the latest versions of most browsers. Earlier browsers will report an error (“Expected ‘;’” or similar) on lines with “let”, so use only where you know which browsers will be used. To see browser support see: http://caniuse.com/#search=let

### Hoisting

#### Variables

Variables created without “var” are automatically created at the Global scope, i.e. as properties of the window object.



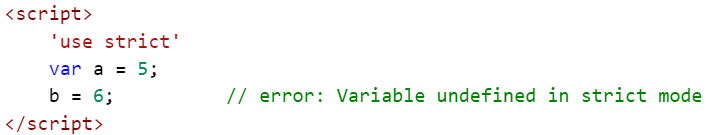
***Hoisting of improperly declared variables is one of the most common errors for new developers. Always declare variables using “var”.***

#### Functions

Default declarations of function create them at the Global scope, as properties of the window object. Exceptions include anonymous functions and functions defined inside of other functions.

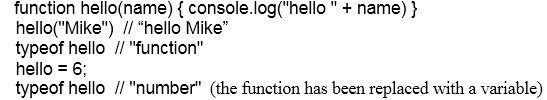
### Strict Mode

To prevent the accidental creation of hoisted variables you can add ‘use strict’ to your code.



* Strict mode can be applied to entire scripts or within individual functions.
* Introduced with ECMAScript 5.
* See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\_mode
* For browser support: http://caniuse.com/#feat=use-strict

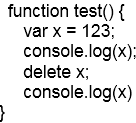
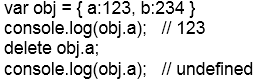
### Creating Variables

* A variable is created at either the Global or function scope when declared using “var”.  
  
* A variable is always created at the Global scope when the first occurrence is an assignment without using “var”.  
  
* Declaring a variable with the same name of an existing variable or function will overwrite the existing item.
  + Example of a variable overwriting a function:  
    
  + Even the built-in functions can be overwritten.  
    

### Deleting Variables

* Variables are deleted when they go out of scope, or when a new page is loaded.
* Unreferenced objects and variables no longer in scope are automatically removed from memory by a garbage collection process.
* Garbage collection is a background process and cannot be forced.
  + How garbage collection in JavaScript works:   
    https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory\_Management
* All variables are discarded when a new page loads.

### The delete Statement

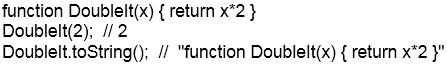
* You cannot not delete a variable declared using “var”.
  + This code:  
      
    returns:  
     123 // before deletion  
     123 // after deletion attempt  
    i.e. “x” was not deleted.
* Properties can be deleted.
  + This code deletes the “a” property:  
    
* Note: You can delete a variable that was “var” declared in a browser’s console. The console uses eval() to run your code.

## Functions

Functions or Methods? Each programming language defines these terms a little differently.

* Often a “function” is defined as a standalone block of code with a name while a “method” is part of an object.
  + C# and Java only have methods.
  + C only has functions.
  + In “Functional Programming” a method has side effects and a function does not. (A function has no side effects when it returns same value, and makes no external changes, every time it is called with the same parameters.)
* In JavaScript, every function is a property of some object, either an object you created or the Global object (window). So, they could be methods.
  + In JavaScript, they are named “function”, so we will call them functions in this course.
  + Just to add to the confusion… functions are not only properties of objects, they are objects!

### Function FAQ

* A function is a JavaScript object and is stored as a variable.
* The function’s object has a .length property which is the number of the object’s parameters.
* The text of a function can be returned using .toString().  
  
* A function can be created from a string.
  + Example with 1 parameter:  
    
  + Example with 2 parameters:  
    

### Anonymous Functions

An anonymous function is simply a function without a name. Anonymous functions are stored in variables or directly assigned to an object’s events.

Not an anonymous function: (it has a name in the function definition!)



An anonymous function stored to a variable:



**The difference?** If created at the Global scope, both of the above are the same. If created at a function scope, the named function is “hoisted” to the Global scope while the function assigned using “var” is at the function scope.

An anonymous function assigned to an event:



### Closures

One of your design goals should be to avoid Globally scoped variables. If your code uses a global variable named “counter”, and another developer’s code also depends on a global variable with the same name, the two blocks of code will “collide” to create some hard to debug errors. The same applies to functions. If your code includes a function named “createlist()”, and another library is loaded that includes a function with the same name, your code could end up calling the wrong function.

The concept of Closure provides both state and privacy. (I.e., no global variables or functions.) In JavaScript closure is achieved by wrapping all of the variables and functions inside of another function. This creates a local scope and prevents name collisions. To provide access to these internal components the Closure function returns not a value, but another function.

More on Closures:

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures
* https://www.w3schools.com/js/js\_function\_closures.asp
* https://www.sitepoint.com/javascript-closures-demystified/

Samples:

Sample File: Module5\Demos\Closure.html

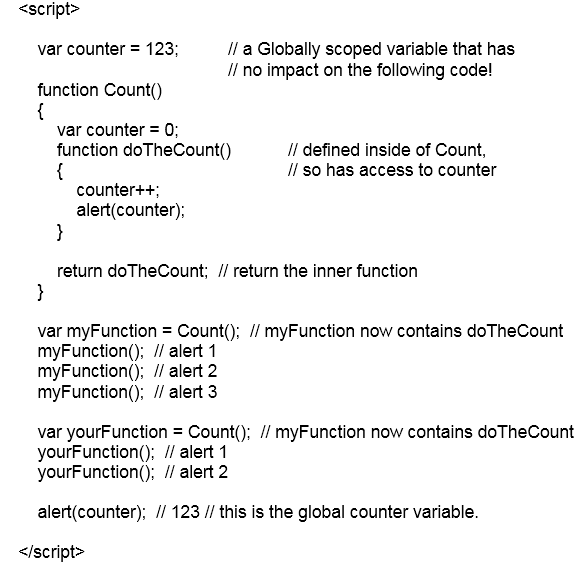
Sample File: Module5\Demos\Closure2.html

#### Example 1:

This example simply wraps a counter variable and a function that increments it.

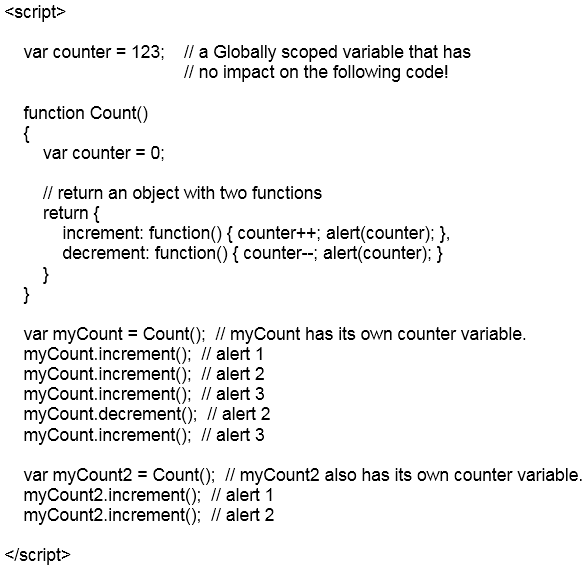
Notes:

* The variable “counter” is defined inside of a function, therefore does not have a global scope.
* When the variable “myFunction” is created, it receives a copy of “doTheCount” *and doTheCount’s environment*, including the local variable “counter”.
* Each time a new function is created from “doTheCount”, a new environment is created. I.e. Each instance will have its own “counter” variable.
* The only object added to the Global scope is the outer function, “Count”.



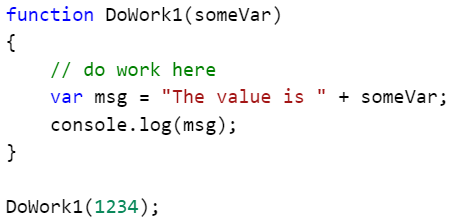
#### Example 2:

This example encapsulates a private variable and two private functions, and exposes the functions as methods (properties) of a returned object.

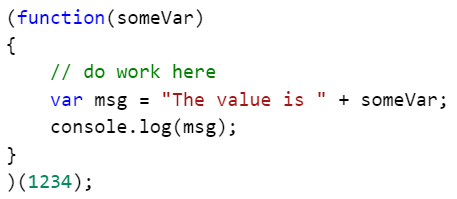


### Self-Invoking / Immediately Invoked Functions

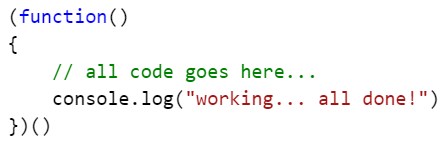
When it’s important that you do not add **any** new objects at the Global scope, you can create a Self-Invoking function. When creating a normal function, you can wrap both variables and functions and only place one object on the global scope, the function itself.



A self-invoking function wraps an anonymous function in parentheses and calls it with “()” added to the end. The function object is created, called and then disposed, never being added to the Global scope. In the example below, neither “someVar” or “msg” are added to the Global scope.



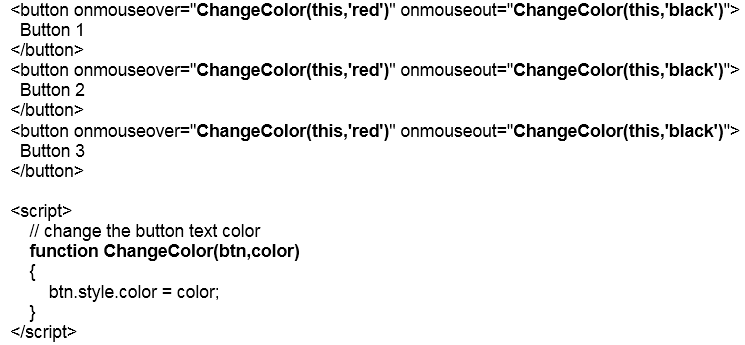
Entire programs can be wrapped this way so as to not add any Globally scoped variables or functions.



## The “this” Keyword

The “this” keyword refers to the owner of the function where the keyword is used, or if not in a function, then it refers to the window object. The most common use of “this” is to pass an element to a function. This will let you create a single function that can be used with many elements.

By using “this”, only one function is needed to change the colors of multiple buttons. In the example below, there are six calls to the ChangeColor function. The function knows which button to change by using the “this” object. No IDs or DOM searches are needed!



The value of “this” varies based on where it is used. See the example code below.

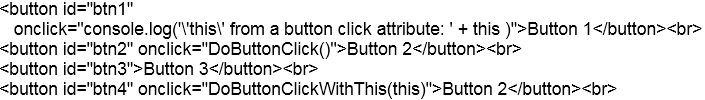
* At the Global scope, “this” refers to window.
* In a Globally scoped function “this” refers to window.
* In a function copied into an element’s event handler “this” refers to the element.
* In an anonymous function stored in an element’s event handler “this” refers to the element.
* In a function called from an event handler “this” refers to window.
* In a function called from an event handler, where the function is called with “this” as a parameter, “this” refers to the element.

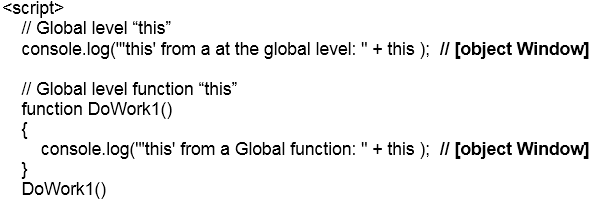
Sample File: Module5\Demos\this.html

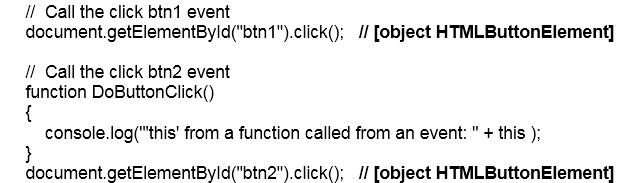
(Load this page with the F12 Console open.)

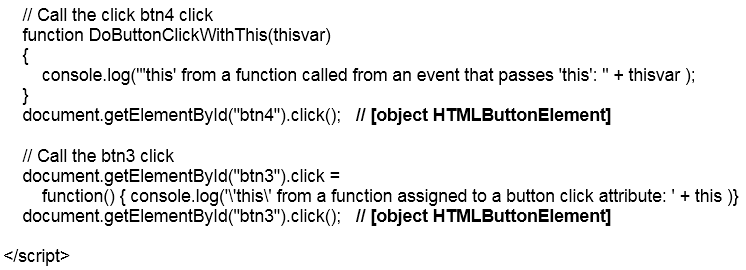
The following code shows examples of “this” access in various ways:

* Inline code on a button click attribute.
* The default global scope value of “this”.
* The default global scope value of “this” from a globally scoped function.
* The value of “this” from an inline function called from a button click attribute.
* The value of “this” from a function called from a button click attribute.
* The value of “this” from a function called from a button click attribute where “this” is passed as a variable.
* The value of “this” from runtime added function called from a button click attribute.

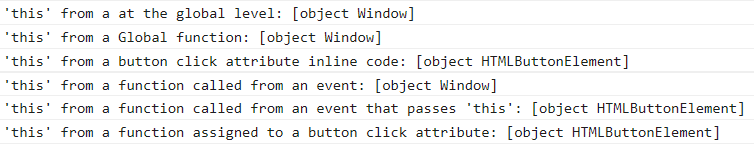








Output from the above code as displayed in the browser’s Console:



## Errors

There are two types of errors: syntax errors and unexpected runtime errors. Syntax errors should be caught by your editor and your testing. Runtime errors are those that are out of your control. These include unexpected user input and slow or non-responding web services.

* If an error is predictable, write a test for it.
  + A user typed “lots” into the Quantity To Order text box.   
    
  + Some browsers do not support a certain property or method. (Missing methods are treated as “false”.)  
    
* When using jQuery AJAX functions, take advantage of their error handling features.

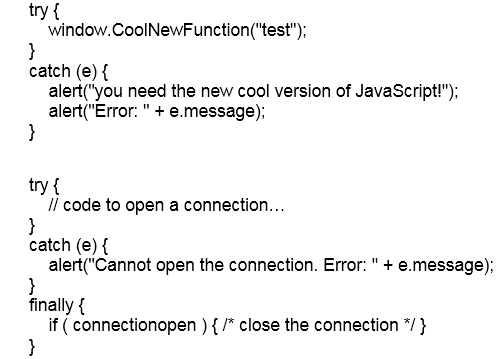
### Try Catch Finally Blocks

When you cannot easily write a test to see if data is valid, you can wrap the code in a try…catch block. This structure has three parts:

* try { … } the code to run
* catch (e) { … } the code to run if the try block raises as error.
* finally { … } (optional) the code to run if the try succeeded or failed. This is the “clean up” block.

Sample File: Module5\Demos\trycatch.html

Examples:

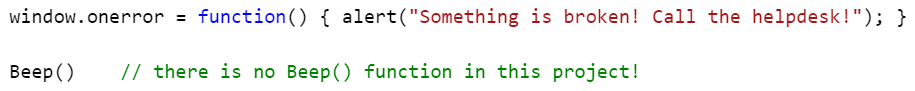
}

Notes:

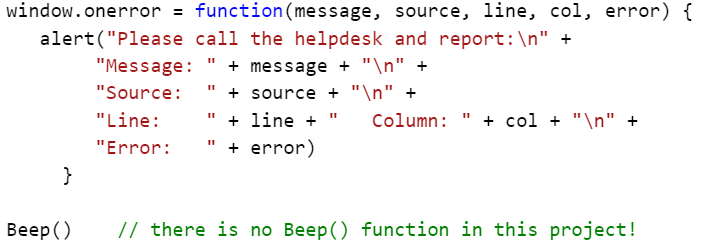
* You can have a try…catch, a try…finally or try…catch…finally.
* In most programming languages, a divide by zero will raise an error. In JavaScript, the same code will execute without error and return a value of Infinity.

### The window.onerror Function

To deal with an error that is not trapped anywhere else in your code you can add a custom function to the window.onerror event handler. This is in effect, the error trap of last resort. The code should appear in your page before any other code.



A better version:



You could also call a custom web service to log the error.

Also see: https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onerror

## Recursion

Simply put, recursion is a programing pattern where a function calls itself until an end condition is reached. As almost every recursive function can be rewritten using a while loop, the choice usually comes down to the data and the clarity of the solution. Recursive solutions are often clearer to read.

Examples of use:

* Calculating factorials.
* “Walking” XML trees, or the DOM.

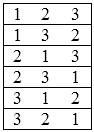
Notes:

* Recursion is not an endless loop. There must be a planned for exit.
* Recursion can be memory intensive as each recursive call adds to the stack.
* JavaScript has limits on the number of recursive calls. Each browser has different limitations and varies from 1000 to over 20000.
  + When that limit is reached you may see:
    - “Maximum call stack size exceeded”
    - “too much recursion”
* Many recursive functions can be rewritten as a while loop.
* Depending on the browser and the function, loops can be many times faster than recursion. (This is mostly due to the stack overhead.)

### A Factorial Example

A factorial is the product of a number and all integers below it. For example, 6! = 6 \* 5 \* 3 \* 2 \* 1. What’s it good for? One use is to determine the maximum number of ways you can arrange a series of numbers.

3! = 3 \* 2 \* 1 which equals 6, so there are six different ways of arranging a collection of three things.



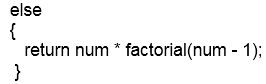
We need a function that accepts a number:



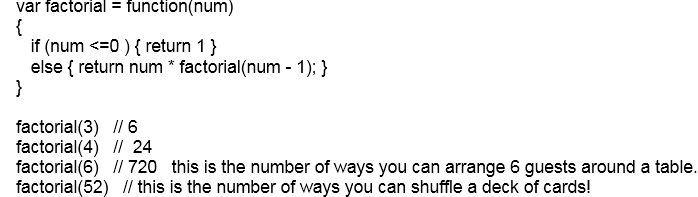
We need a test to see if we are done and an exit:



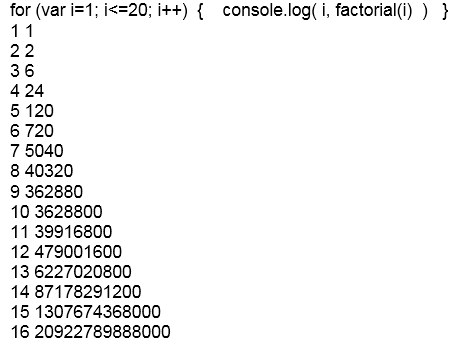
And if not done we need to multiply the number by one less than itself. ( 3 \* 2 \* 1 );



Here’s the function and a use of it:



And if you call it many times:



As there is always more than one way to solve a problem, here’s a factorial function written using a while loop.

