Week 04 - Unobtrusive JavaScript

**JS FUNCTIONS (DEFINITIONS):**

\*\*Defined using the ‘function’ keyword \*\*Can be used as values and in expressions

Function Declaration:

* Not executed immediately, but when invoked
* Not an executable statement 🡪 no ‘ ; ’

function myFunction(a, b) { return a \* b; }

Function Expression:

* Can be stored in a variable & that variable can be used as a function
* Is an executable statement 🡪 ends with ‘ ; ’

var x = function(a, b) {return a \* b};  
var z = x(4, 3);

* Functions stored in variables do **not need a function name** (anonymous)
* They are always invoked (called) using the variable name

Function() Constructor:

* Functions can also be defined using the built-in JS function constructor

var myFunction = new Function("a", "b", "return a \* b");

* Unnecessary to use the function constructor; same as writing:

var myFunction = function (a, b) {return a \* b}; //Usually can avoid ‘new’ in JS

Function Hoisting:

* JS’s default behavior of moving declarations to top of current scope; variable & function
* Functions defined using an expression are not hoisted.

Self-Invoking Functions:

* Function expression can be made self-invoking – executed without being called
* Will execute automatically if followed by ‘()’
* Add ‘( )’ around function to indicate that its an expression

Functions Are Objects:

\*\*typeof returns functions as ‘function’ BUT they’re best described as Objects

* JS Functions have both:
* Properties: ex 🡪 arguments.length
* Methods: ex 🡪 toString()

\*A Function defined as the property of an object 🡪 method to the object

\*A Function designed to create new objects 🡪 object constructor

**JS FUNCTION PARAMETERS:**

\*\*Parameters: names listed in function definition

\*\*Arguments: real values passed to & received by the function

* Rules: JS Functions do not:
* Specify data types for parameters
* Perform type checking on the passed arguments OR check number of arguments received
* Defaults:
* If a function is called with missing arguments 🡪 missing values set to undefined
* Can assign a default value for the parameter in function

function myFunction(x, y) {  
    if (y === undefined) {  
          y = 0;  
    }   
 }

Arguments Object:

* Built-in object that contains an array of the arguments used when the function was called
* Allows you to use a function to find: ex – sum of all input values/arguments

\*\*Arguments:

* Passed by Value
* Changes to arguments are not visible (reflected) outside the function

\*\*Objects:

* Passed by Reference
* Changes to Object Properties are not visible (reflected) outside the function

**JS FUNCTION INVOCATION:**

**Invoking a JS Function:**

* Function is not executed when declared; must be invoked (called)

**Invoking a Function as a Function:**

\*\*In JS there is always a default Global Object 🡪 the HTML page itself

\*\*In a web browser the Page Object is the Browser Window

function myFunction(a, b) {  
    return a \* b;  
}  
myFunction(10, 2); / window.myFunction(10,2);

* This Function automatically becomes a Window Function;

myFunction() & window.myFunction are the same

The **this** Keyword:

\*\*In JS the thing called ‘this’ is the object that owns the current code

**The Global Object:**

* \*\* When a function is called without an owner object, the value of **this** becomes the global object
* \*\*In a web browser the global object is the browser window

function myFunction() {  
    return this;  
} //This Function returns the Window Object as the value of **this**  
myFunction();   **\*\***Using the Window Object as a variable can crash the program

**Invoking a Function as a Method:**

* \*\*In JS you can define functions as Object Methods

var myObject = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function () {  
        return this.firstName + " " + this.lastName;  
    }  
}  
myObject.fullName(); //myObject owns the fullName Function

\*\*Invoking a Function as an Object Method causes the value of **this** to be the object itself

**Invoking a Function with a Function Constructor:**

* \*\*If a function invocation is preceded with the **new** keyword, it is a constructor invocation

// This is a function constructor:  
function myFunction(arg1, arg2) {  
    this.firstName = arg1;  
    this.lastName  = arg2;  
}  
  
// This creates a new object  
var x = new myFunction("John", "Doe");  
x.firstName;

\*\***this** doesn’t have a value in a constructor 🡪 the value of **this** will be the new object created when the function is invoked

**JS HTML DOM (Document Object Model)**

\*\*A platform & language neutral interface that allows programs and scripts to dynamically access & update the content, structure, & style of a document

\*\*The HTML DOM model is a tree of objects; starting with document

* W3C DOM Standard is separated into 3 parts:
* HTML DOM: standard model for HTML documents
* XML DOM: standard model for XML documents
* Core DOM: standard model for all documents

**HTML DOM:** standard for how to get, change, add, or delete HTML elements

* Standard object model & programming interface for HTML that defines:
* HTML elements as objects
* Properties of elements
* Methods to access elements
* Events for elements
* DOM Programming Interface: The Properties & Methods of each object

\*\*Can be accessed with JS and other languages

\*\*In the DOM all HTML elements are defined as objects

* Methods: actions you can perform on HTML elements

-getElementById: accesses an element

* Property: value you can set or get

-innerHTML: gets content of an element

DOM **Document Object:** represents the webpage

* To access any elements, start with this
* Finding elements:
* Changing elements:
* Adding/Deleting elements:
* Adding events handlers:
* Finding objects: \*www3schools objects, collection, & property table\*

**ELEMENTS:**

* Find HTML elements by
* id: document.getElementById(“id”)
* class name: document.getElementsByClassName(“class name”)
* tag name: document.getElementByTagName(“p”) 🡪 finds all p elements
* CSS selectors: querySelectAll() – finds all elements that match a specified CCS selector

Ex. document.quertSelectorAll(“p.intro”) 🡪 returns all p elements with the class ‘intro’

* JS DOM HTML:
* Changing HTML content:

document.getElementById(‘id’).innerHTML = new HTML

* Changing HTML attributes:

document.getElementById(‘id’).attribute = new value

* JS DOM CSS:
* Changing an HTML style:

document.getElementById(‘id’).style.property = new style

**EVENTS:**

* Events are generated by the browser when “things happen” to HTML elements
* An element is clicked
* The page has loaded
* Input fields are changed
* Visibility:

onclick="document.getElementById('id').style.visibility= “hidden/visible”

\*HTML DOM Style Object Reference 🡪 for all style properties\*

Element content changes when user clicks on it

onclick = “this.innerHTML” = ‘new content’

Function is called from event handler

HTML 🡪 <h1 onclick="changeText(this)">Click on this text!</h1>

JS 🡪 function changeText(h1) { h1.innerHTML = "Ooops!"; }

Event Attributes: used to assign events to HTML elements

<button onclick = “displayDate()”> 🡪 This function will execute when button is clicked

Assign events using the DOM/ JS

document.getElementById(myButton).onclick = displayDate();

* onload & onunload Events:
* Triggered when user enters/leaves page
* Used to check visitor’s browser type & version, and to deal with cookies

<body onload = “checkCookies()”>

* onchange Event:
* Often used in combination with validation of input fields
* onmouseover & onmouseout Events:
* Triggers function when user moves mouse over or out of an element
* onmousedown & onmouseup Events:
* Triggered when mouse is clicked/released

\*HTML DOM Event Object Reference- list of all events\*

**EVENT LISTENERS: (**handler & listener are synonymous**)**

* addEventListener() method
* Attaches an event handler to specified element w/out overwriting existing ones
* Write many events to the same element w/out overwriting existing ones
* Add event listeners to any HTML DOM object (document, window) & objects that support events
* Syntax: element.addEventListener(event, function, useCapture)

Ex. document.getElementById(‘id’).addEventListener(“click”, displayDate);

( DON’T USE THE “ON” PREFIX FOR THE EVENT )

* Alert “Hello World!” when the user clicks on an element:

element.addEventListener("click", function(){ alert("Hello World!"); });

* Refer to an external named function:

element.addEventListener("click", myFunction);  
 function myFunction() {  
    alert ("Hello World!"); }

\*WHEN PASSING PARAMETERS: use anonymous function that calls the specified function w/ params

element.addEventListener("click", function(){ myFunction(p1, p2); });

**EVENT PROPAGATION:**

* Way of defining the element order when an event occurs; two ways:
* Bubbling: the inner most element’s event is handled first
* Capturing: the outer most element’s event is handled first
* \*Use the addEventListener() method to specify ‘useCapture’ parameter
* True: bubbling ; False: capturing
* removeEventListener() method:
* removes event handlers attached with the addEventListener() method

**DOM NODES:**

* This code creates a new <p> element:

var para = document.createElement("p");

* To add text to the <p> element, you must create a text node first. This code creates a text node:

var node = document.createTextNode("This is a new paragraph.");

* Then you must append the text node to the <p> element:

para.appendChild(node);

* Finally you must append the new element to an existing element.
* This code finds an existing element:

var element = document.getElementById("div1");

* This code appends the new element to the existing element:

element.appendChild(para); 🡪 appends new element as child of the parent OR

element.insertBefore(para, child);

**REMOVING EXISTING HTML ELEMENTS:** must know the parent element

* This HTML document contains a <div> element with two child nodes (two <p> elements):
* <div id="div1">  
  <p id="p1">This is a paragraph.</p>  
  <p id="p2">This is another paragraph.</p>  
  </div>
* Find the element with id="div1":
* var parent = document.getElementById("div1");
* Find the <p> element with id="p1":
* var child = document.getElementById("p1");
* Remove the child from the parent:
* parent.removeChild(child);
* \*Find the child you want to remove, and use its parentNode property to find the parent:
* var child = document.getElementById("p1");
* child.parentNode.removeChild(child);

**REPLACING HTML ELEMENTS:**

* Use the replaceChild() method

\*\*\*Missing DOM Nodes & DOM Node List Notes\*\*\*

**Unobtrusive JavaScript Explained**

David Flanagan's seminal *JavaScript: The Definitive Guide* says that while there is no specific formula, there are three main goals:

* To separate JavaScript from HTML markup, as well as keeping modules of JavaScript independent of other modules.
* Unobtrusive JavaScript should degrade gracefully - all content should be available without all or any of the JavaScript running successfully.
* Unobtrusive JavaScript should not degrade the accessibility of the HTML, and ideally should improve it, whether the user has personal disabilities or are using an unusual, or unusually configured, browser.
* Unobtrusive JavaScript should add as little as possible to the global [object](https://en.wikipedia.org/wiki/Object_(computer_science)) or global [namespace](https://en.wikipedia.org/wiki/Namespace) of the environment in which it runs
* Writing an event listener that detects the loading of the HTML page and then adds relevant listeners to other events on the page, as well as other behaviors as required, can solve the problem of separating JavaScript functionality from HTML markup. The use of client-side JavaScript libraries such as [jQuery](https://en.wikipedia.org/wiki/JQuery), [MooTools](https://en.wikipedia.org/wiki/MooTools) or [Prototype](https://en.wikipedia.org/wiki/Prototype_JavaScript_Framework) can simplify this process and help ensure that individual browser and browser version implementation details are [hidden](https://en.wikipedia.org/wiki/Abstraction_(computer_science)) and catered for. Keeping most of the JavaScript out of the default namespace helps ensure that it is as unobtrusive as possible in that sense. A further criterion of unobtrusive JavaScript that is often cited is to ensure that added behavior degrades gracefully on those browsers with unexpected configurations, and those on which the user may have disabled JavaScript altogether.
* This way, all the client-side functionality depends on the same JavaScript function. If that function succeeds, it begins by removing the basic, manual behavior, and goes on to add the client-side scripted behavior. If the script fails for whatever reason, the manual behavior remains in place and remains functional.

Though the essence of unobtrusive JavaScript is the concept of an added separate behavior layer, advocates of the paradigm generally subscribe to a number of related principles, such as:

* [DOM Scripting](https://en.wikipedia.org/wiki/DOM_Scripting), i.e. adherence to the [W3C](https://en.wikipedia.org/wiki/World_Wide_Web_Consortium) [DOM](https://en.wikipedia.org/wiki/Document_Object_Model) and event model, and avoidance of browser-specific extensions.
* [Capability detection](https://en.wikipedia.org/wiki/Progressive_enhancement), i.e. testing for specific functionality before it is used.[[18]](https://en.wikipedia.org/wiki/Unobtrusive_JavaScript#cite_note-18) In particular this is seen as the opposite of browser detection.
* More generally, JavaScript best practices often parallel those in other programming languages, such as [encapsulation](https://en.wikipedia.org/wiki/Information_hiding) and [abstraction layers](https://en.wikipedia.org/wiki/Abstraction_layer), avoidance of [global variables](https://en.wikipedia.org/wiki/Global_variable), meaningful [naming conventions](https://en.wikipedia.org/wiki/Naming_conventions_(programming)), use of appropriate [design patterns](https://en.wikipedia.org/wiki/Design_patterns), and systematic [testing](https://en.wikipedia.org/wiki/Software_testing). Such principles are essential to large-scale software development, but have not been widely observed in JavaScript programming in the past; their adoption is seen as an essential component of JavaScript's transition from a "toy" language to a tool for serious development.