**JAVASCRIPT**

***Ctrl + Shift + C*** - Keyboard Shortcut to Open the “Inspect” Window.

Changing the Image on an image tag in a web page

***document.querySelector(‘img’)*** – Selects the image html tag on the web page.

***const ourImage = document.querySelector(‘img’);*** Places the image obtained into a variable named ”ourImage”.

***ourImage.setAttributes(‘src’, ‘https://imageio.forbes.com/specials-images/imageserve/5f962df3991e5636a2f68758/0x0.jpg?format=jpg&crop=812,457,x89,y103,safe&width=1200’)***

*– Changes the image source for the image tag stored in “src” with “…”.*

***ourImage.setAttributes(‘width’, 100)*** – Changes the width of the image specified by “ourImage” to 100 pixels.

***clear()*** *– Clears the console.*

**Function xyz(133){**

**}** - Syntax for declaring a function where “xyz” is the function name and “123” is the argument.

Changing the colour of the web page after clicking anywhere within it

***const htmlBody = document.querySelector(‘body’);***

***const randomClickFunction = function () {***

***const colors = [”red”,”orange”,”yellow”,”green”,”blue”,”purple”,”black”,”grey”];***

***const randomIndex = Math.floor(Math.random() \* colors.length);***

***randomColor = colors[randomIndex];***

***htmlBody.style.background = randomColor;***

***console.log(‘The User Clicked and Changed the Color to ’ + randomColor + ‘.’);***

***}***

***htmlBody.onClick = randomClickFunction;***

Variables and Data Types

***let anotherVariable;*** - Declares a variable called “anotherVariable”. This variable can be re-assigned/re-initialised.

***const firstVariable* = 20;** - Creates a variable called “firstVariable” and initializes it to “20”. This variable cannot be re-assigned.

***var variable1* = 20;** - Creates a variable called “variable1” and initializes it to 20. This variable can, however, be re-declared.

***anotherVariable = 20;*** - Initializes the variable declared above.

***console.log(‘Hello World’);*** – Prints “Hello World” to the console.

***console.log(firstVariable);*** – Print 20, the value of the “firstVariable” variable.

***typeof variable1;*** - Returns the type of the variable named “variable1”.

***Number(variable1)*** – Explicitly casts the value placed inside “variable1” variable into a number.

String Functions

***myString.split(****separator****)*** – Splits the string called “myString” into an array. (E.g if myString = “Hello World”:

*myString.split(‘’) returns [“H”,”e”,”l”,”l”,”0”,” ”,”W”,”o”,”r”,”l”,”d”]*

*myString.split(” ”) returns [“Hello”,”World”]*

*myString.split() returns [“Hello World”]*

***myString.split(****separator, limit****)*** – Splits the string called “myString” into an array. (E.g if myString = “Hello World”;

*myString.split(‘’) returns [“H”,”e”,”l”,”l”,”0”,” ”,”W”,”o”,”r”,”l”,”d”]*

*myString.split(” ”) returns [“Hello”,”World”]*

*myString.split() returns [“Hello World”]*

***myString.trim()*** – Removes the white spaces at the start and end of a string and returns this shadow string. E.g. if myString = ‘ Hello World ’;

*myString.trim() returns ‘Hello World’;*

*myString.trimStart() <or trimLeft() - alias> returns ‘Hello World ’;*

*myString.trimEnd() <or trimRight() - alias> returns ‘ Hello World’;*

***myString.repeat(n)*** – Repeats “myString” n times. E.g. if myString = “abc”

*“myString.repeat(3)” returns “abcabcabc”*

***myString.replaceAll(substring|regex, newSubstring)*** – Replaces a substring or character sequence, within a larger string, with some other string. For example,

*const myString = “my Dog jumped on the bed. My dog is a bad Dog”;*

*const newStringA = myString.replaceAll(‘dog’, ‘cat’);*

assigns to newStringA; “my Dog jumped on the bed. My cat is a bad Dog”.

N/B: replaceAll is case sensitive hence, the “Dog” is not replaced.

*Const newStringB = myStringB.replaceAll(/[Dd]{1}og/g, “cat”);*

Assigns to newStringB; “my cat jumped on the bed. My cat is a bad cat.”

Here: the regular expression matches a string that is as follows:

Starts with either a “D” or a “d”, specified by the {1}

Is followed by “og”, so “Dog” or “dog”

Is anywhere in the larger string, specified by the “/g”, the global flag.

***myString.toUpperCase()*** – Capitalises whatever string is passed on to it. E.g.

*const myString = “some string”;*

*myString.toUpperCase();* - returns “SOME STRING

myString.substring(startIndex, [finishingIndex]); - returns the part of “myString” from the start index to the index before the second one specified. E.g.

*const myString = “some string”;*

*myString.subString(5);* - returns “string”;

*myString.subString(0, 4);* - returns “some”;

Array Functions

N/B – Arrays are mutable unlike primitives such as string, which are immutable (cannot be changed).

***const myArray* = [10,20,30,40];** - Creates an array called “myArray” and adds 10, 20, 30 and 40 and indices 0 to 3.

***myArray.push(50,60,70);*** – Adds 50,60 and 70 to the end of “myArray” array.

***myArray.unshift()*** - Adds elements to the beginning of an array.

***myArray.pop()*** – Removes the last element from an array and returns it.

***myArray.shift()*** – Removes the first element of an array and returns it.

***myArray.reverse();*** – Reverses the order of the array elements with the “myArray” array.

***myArray.join(****separator, e.g. <‘’, ‘,’, ‘-’>****)*** – Joins the array called “myArray” into an string.

***myArray.slice(x1,x2)*** – Split “myArray” into another shadow array containing the elements from index “x1” to the one just before index “x2”. E.g if myArray = [1,2,3,4,5,6],

*myArray.slice(2) returns [3,4,5,6]*

*myArray.slice(-2) returns [5,6]*

*myArray.slice(2,4) returns [3,4]*

*myArray.slice(2,-2) returns [3,4]*

***myArray.splice(startIndex, deleteCount, newElement1,…)*** – Deletes a number of elements, specified by “deleteCount” from “myArray” starting from index “startIndex”, and places the “newElement”s starting at that “startIndex”. It then returns the elements removed. E.g.

If *myArray = [1,2,3,”oops”,”haiya”,7,8,9];*

myArray.splice(3,2,4,5,6); - Removes “oops” and “haiya” and places “4,5,6” starting from the same startIndex. It, however, returns [“oops”, “haiya”]

***myArray.indexOf(someElement)*** – Returns the index of some element in the array. E.g.

If *myArray = [1,2,3,4]*

*myArray.indexOf(3) – returns 2, the index of “3”.*

***myArray.findIndex(Callback Fn., thisArg)*** – Returns the index of the first element that meets a certain criteria, specified in the callback function. If no element fit the criteria, it returns -1. E.g.

Given an array of objects:

*const complexArray = [{firstName: ‘Bob’,*

*LastName: ‘Smith’},*

*{firstName: ‘Alice’,*

*LastName: ‘Smith’},*

*{firstName ‘Jon’,*

*LastName: ‘Smith’},*

*{firstName: ‘Jon’,*

*LastName: ‘Doe’}]*

*moreComplexArray.findIndex((arrayItem) => {return arrayItem.lastName !== ‘Smith’;}) – returns 3, “Jon Doe” the forth object in the array.*

***myArray.map()*** – Creates a new array whose individual elements are results of a function specified. It simply performs a transformation of an array. E.g. if myArray is [1,4,9,16]

*const array1 = myArray.map((num) => Math.sqrt(num)) - returns [1,2,3,4]*

***myArray.forEach(callback Fn., Index)*** - Similar to writing a for loop but in a more concise manner. E.g.

If *myArray = [1,2,3,4];*

A for loop that displays all the elements in the array, as well as their indices would be:

*for (let i=0, i<myArray.length; i++){*

*console.log(myArray[i]);*

*console.log(i);*

*}*

This loop can be summarised as follows

*myArray.forEach((arrayItem, index) => {console.log(arrayItem);*

*console.log(index)});*

This displays the same thing.

***myArrray.includes(element)*** – Returns true if the “element” is a member of “myArray”. E.g.

If *myArray = [“apple”, “mango”, “banana”, “orange”, “pear”];*

*myArray.includes(“orange”) - returns true.*

***array1.concat(array2, array2)*** – Merges two or more arrays. E.g. if array1 = [1,2,3], array2 = [4,5,6] and array3 = [7,8,9]

*array1.concat(array2, array2) returns [1,2,3,4,5,6,7,8,9]*

***myArray.filter(Callback Fn.)*** – Returns the elements that meet a certain criteria. Similar to “findIndex()” except it returns the actual elements rather than the index. E.g.

If *allOrders = const allOrders = [*

*{*

*ProdctName:'Tea Pot',*

*isDigital:false,*

*isCancelled:false,*

*isOpen:false*

*},*

*{*

*ProdctName:'Blue Gildan Men"s Hoodie',*

*isDigital:false,*

*isCancelled:true,*

*isOpen:false*

*},*

*{*

*ProdctName:'Code Complete Kindle Book',*

*isDigital:true,*

*isCancelled:true,*

*isOpen:false*

*},*

*{*

*ProdctName:'Atomic Habits Kindle Book',*

*isDigital:true,*

*isCancelled:false,*

*isOpen:false*

*}*

*];*

To filter out all the items that are digital:

*allOrders.filter((arrayItem) => {return arrayItem.isDigital === true}); - This returns:*

*{*

*ProdctName:'Code Complete Kindle Book',*

*isDigital:true,*

*isCancelled:true,*

*isOpen:false*

*},*

*{*

*ProdctName:'Atomic Habits Kindle Book',*

*isDigital:true,*

*isCancelled:false,*

*isOpen:false*

*}*

***myArray.reduce(callback Fn., initialValue)*** – Execute the “callback fn.” On each element of “myArray”. Here, an element is taken as an argument to this function, worked on, and the result used as the other argument when the next element in “myArray” is taken up. Note: The final result of the reduce method is a single value.

Eg. If myArray = [1,2,3,4] and the callback Fn. Is as function sumOfNumbers (x,y) {return x + y},

*myArray.reduce(sumOfNumbers, 10) returns 20; //((((10+1 = 11)+2 = 13)+3 = 16)+4 = 20)*

***myArray.sort()*** – Sorts “myArray” in ascending order, changing “myArray” itself. If not intending to change “myArray”, use ***myArray.toSorted().***  E.g.

If *myArray = [5679213432567428]*

*numFinal = Number(String(myArray).split('').sort().join('')); - is assigned 1222334455667789 as a number*

Objects

***const objectVariable = {property1: 20, property2: ‘Hello’};*** - Creates an object named “objectVariable” and declares variables, property1 and property2, within it whose values are “10” and “Hello” respectively. This is what is called a Key-Value pair, where, for example, “property1” is the key and 20 is the value.

***objectVariable.property1;*** – Returns 20, the value of “property1” key within “objectVariable” object. This is what is called dot notation.

***objectVariable[‘property1’];*** – Also returns 20. This is an alternative to the dot notation.

***Const objectVariable {layer1: {layer2: {layer3: {targetValue: 20}}}};*** – Creates a nested object. Here:

**objectVariable.layer1.layer2.layer3.targetValue;** *–* Returns 20. Dot notation.

**let objectKeys = Object.keys(someObject)** – Creates an array called “objectKeys” and assigns, as its values, the keys (in the key:value pair) of the object “someObject” passed onto it.

**navigator** – Vanilla object that returns the users geographical and machine information

*navigator.geolocation.getCurrentPosition((position) => {*

*console.log(position.coords.latitude); // Displays the latitude of the device*

*console.log(position.coords.longitude); // Displays the longitude of the device*

*})*

Operators

**\*\*** - Exponential *e.g. 8\*\*2 = 64*

*Do not work with Arrays and Objects*

**===** - Equal in both value and type *(e.g. 20 === ‘20’ evaluates to false)*

**!==** - Not Equal in both value and type *(e.g. 20 !== ‘20’ evaluates to true)*

**==** - Equal in Value *(e.g. 20 == ‘20’ evaluates to true).*

**!=** - Not Equal in value *(e.g. 20 != ‘20’ evaluates to false)*

***const result = 20 === ’20 ? ‘values match’ : ‘values do not match’;*** - Ternary operator. Evaluates 20 === ‘20’ (false). If true, it assigns the string “values match” to variable “result”. If false it assigs the string “values do not match” to variable “result”.

**&&** - Logical AND *(e.g. 20 === 20 && 10 === 12 evaluates to false)*

**||** - Logical OR *(e.g. 20 === 20 || 10 === 12 evaluates to true)*

**!** - Logical NOT *(e.g. !(20 === 13) evaluates to true)*

*Lodash – JS Library for more complex functionalities.*

Functions

***function myFuntion() {***

***console.log(‘Hello World’)***

***}();*** - Standard way to declare a function.

***const anotherFunction = function () {***

***console.log(‘Hello World’)***

***}*** – Another way to declare a function. Note, it has not been given a name. It is, therefore, called an anonymous function. However, it has been assigned to a variable. So to call/invoke/execute the function, you type, ***anotherFunction();***

***const arrowFunction = () => {***

***console.log(‘I am an arrow function’);***

***}*** – Another way to declare a function. Similar to the one above but with the extra “=>” and omitted “function” keyword. It is invoked in the same manner as the one above:

***arrowFunction().***

***(function myFuntion() {***

***console.log(‘Hello World’)***

***})();*** - Immediately Invoked Function. It runs there and then. Note the parentheses before and after the “function” keyword and before the ().

**const aliasVariable = myFunction;** - Assigns an already created variable named “myFunction” to a variable. Note the missing parentheses after “myFunction”. Consequently, it is possible to assign a function to an object as a value to some property. E.g.

***const myObj = {***

***property1: 20,***

***property2: aliasVariable***

***};*** *-* To execute it, ***myObj.property2();*** Note, when a function is attached to an object, it is then referred to as a method.

**Callback Functions** – These are functions that are passed, as arguments, to other functions within which they are called. These allow for function reusability and asynchronous programming. E.g. given two functions:

*function doubler(x){*

*return X\*2;*

*}*

*function callerFunction(someNumber, functionToCall){*

*functionToCall(someNumber);*

*}*

If the following arguments were passed to “callerFunction”:

*callerFunction(20,doubler);*

A return of 40 would be made.

Other Functions

***valueOf*** – E.g if string1 = ‘Hello World’

*string1.valueOf() returns ‘Hello World’*

Date Functions

myDate = Date.now() – Assigns the value of number of milliseconds that have elapsed since the midnight of January 1, 1970 to “myDate”. Equivalent to myDate.valueOf()

Given:

*const myDate = new Date(); - Conventional way of declaring a date variable. Assigns the current date to “myDate”.*

*myDate.toString()*

*myDate.toISOString()*

*myDate.getDate()*

*myDate.getMonth()* – Returns the month but the value as an index. So January is returned as 0, and December as 11.

*myDate.getFullYear()*

*myDate.valueOf* – Returns the number of milliseconds that have elapsed since the midnight of January 1, 1970.

myDate.toDateString – Casts the date to a string

Regular Expressions

Declarations:

*const myRegex = new RegExp(‘favorite’)*

*/Favorite/*

For both the above, whatever string will be passed will be checked to see if the character sequence “favorite” is there. This checking is carried out as follows:

If:

*const string1 = “my favorite book”;*

*const string2 = “my FAVORITE book”;*

then:

*myRegex.test(string1) – returns true;*

*myRegex.test(string2) – returns false;*

or

*/favorite/.test(string1) – returns true;*

*/favorite/.test(string2) – returns false;*

Identifiers:

[] – Groups of Characters. E.g.

*/[A-Za-z0-9 ]/* - checks if the string passed has any uppercase or lowercase characters, or numbers, or spaces.

Period(.) – Any character, or sequence of characters whatsoever

\d – any digits (equivalent to [0-9])

\s – Any white spaces.

\w – Any alphanumeric character (equivalent to [*A-Za-z0-9*])

Caret(^) – The beginning of the string. E.g.

*/^m/.test(string1)* - returns true because the first character in string1 is “m”.

*/^f/.test(string1)* - returns false because the first character in string1 is not “f”.

- Not containing , Eg.

*If myString = “1234”,*

*/^A-Za-z/.test(myString) returns true because “myString” contains no alphabetic characters*

Dollar sign($) - The end of the string. E.g.

*/k$/.test(string1)* - returns true because the last character in string1 is “k”.

*/h$/.test(string1)* - returns false because the last character in string1 is not “h”.

Optional sequences() E.g.

*/(my|cat)/.test(string1)* – returns true because string1 contains the string “my” within it even though “cat” is not there.

Quantifiers:

Often used with the *.exec* method which returns an array of strings that match the regular expression.

\* - Zero or many

+ - one or many

? – zero or one

{range} – E.g. {2,6}

Flags

Check mdn.

Math Functions

N/B: Math functions are static functions rather than instance methods

Math.PI

Math.E

Math.abs()

Math.min()

Math.max()

Math.ceil()

Math.floor()

Math.round()

Math.random()

Error Handling

try{

// Code to be executed if all things go well

} catch (error) {

// Code to be executed if something goes wrong in the “try” code block above.

}

E.g.

try{

const myNum = 50;

myNum.toUpperCase(); // you cannot capitalize a number (type error)

} catch (e) {

Console.log(e instanceof TypeError); /\* “Is error ‘e’ a type error?” - Returns true

console.log(e.message); // Returns (myNum.toUpperCase is not a function)

}

**JSON (Javascript Object Notation)**

Data is sent btw computers in the form of strings.

For this reason, we can change an object into a string for transmission for transmission. This can be done via the:

*JSON.stringify(someObjectVariable).*

After receiving this stringified object, we can convert it back to an object using:

*JSON.parse(someString)*

It is important to note that the device sending the data, in most cases, is the server. This data has to first, be requested for. To do this:

An XMLHttpRequest can be sent. This is done as follows:

A new XMLHttpRequest object is created: *let request = new XMLHttpRequest();*

This created object is opened via a method that accepts three arguments: the operation to perform, the Url to perform this operation at/from, the nature of the operation, whether synchronous (true) or asynchronous (false): *request.open(‘GET’, “someURL”, true);*

When returned, the response will be assigned to the “responseText” field of the XMLHttpRequest.

Before doing anything to the XMLHttpRequest (after the response is returned via the same), it is wise to ensure that the response has in fact been sent because there may be delays. This is done via doing this “anything” within the “onload” property of the XMLHttpRequest such as: *request.onload = () => {console.log(request.responseText)} // Displays the returned data that is in the form of a string.*

The XMLHttpRequest is then sent via: *request.send()*

The response can then be stored in a variable via the JSON methods above:

*let result = JSON.parse(request.responseText)*

This “result” is now a valid javascript object that can be manipulated as you wish e.g. via for loops that get each element of the object

A fetch Method can be used

*let someArray = []; // can be anything.*

*fetch(“url”).then((response) => {response.json()}).then((data => someArray = data;));*

This works as follows:

*fetch()*  - Returns a promise (which has several methods, one of which is then) which has two properties. The status (“Resolved”, “pending”) and the value (the actual data). This value is in the form a “response” object.

#1 *then()* - It is a method on the promise returned. It takes one argument, the value from the fetch() promise returned. This value is in the “response” object above. More specifically, within this “response” object, there is the “body” key/attribute.

This body, however, cannot be worked with directly. For this reason, the json() method is used. It converts this body into a javascript object and returns it as another promise within the value attribute of the returned promise. This however only works if the status of the original promise from fetch is “resolved”.

Also, the reason why the json() method returns a promise as a response is because there may be errors in the conversion, which, can then be flagged on promise status.

#2 *then()* - This now works on the json() response promise and from it takes its value (the actual object) as an argument. You can do whatever you want within this part function. Here, this object is assigned to an already created array variable.

Rendering JSON data on the web page.

Remember this JSON data is in the form of an object. So iteration through the same may be necessary and object-specific methods such as “Object.keys(someObject)” may be necessary. Also, a variable that stores the html as a string will be necessary. With these, you can concatenate whatever data is returned from the above (loops and methods) as well as the html tags surrounded by quotes, e.g.

*let html = “”; // String variable to hold the HTML.*

*jsonObject.forEach((people) => { /\* Looping through the jsonObject which contains details about about many people. \*/*

*html += “<div class=people>”; // For each person creates a div.*

*let personAttributes = Object.keys(jsonObject); /\* Creates an array to store the keys of each person \*/*

*personAttributes.forEach( /\* For each attribute, it renders the key:value pair to html \*/*

*html += “<strong>” + personAttributes + “</strong> : ” + people[personAttributes] + “<br />”;*

*)*

*Html =+ “</div>” // Closes the div created for the individual person.*

*})*

Sending/Posting Data to an external Server

We use the XMLHttpRequest again. Here we work with three properties of the same:

readyState (to tell if the request is finished), Status (to tell whether the POST has been created in the server) and Response (the value created and returned from the server)

readyState === 4 // Done – Only indicated that the operation was completed but not that it was completed successfully. Hence,

status === 201 // Created

The XMLHttpRequest works as follows:

let req = new XMLHttpRequest();

req.open(“POST”, url, true) /\* Post asynchronously to location “url” \*/

req.setRequestHeader(“Content-Type”, “application/json; charset=UTF-8”)

/\* Header: Value pair \*/

Req.onreadystatechanged = () => {

If (req.readyStatus === 4 && req.status === 201){ //If completed and Successful

let object = JSON.parse(req.response);

}

}

let body = JSON.stringify({name: “Daniel”, age: “22”}); // The message has

both a header and a body

req.send(body) // The message has both a header and a body