***JavaScript Notes***

* Hiding HTML elements can be done by changing the display style:
  + document.getElementById("demo").style.display = "none";
* Showing hidden HTML elements
  + document.getElementById("demo").style.display = "block";
* Old JavaScript examples may use a type attribute: <script type="text/javascript">.  
  The type attribute is not required. JavaScript is the default scripting language in HTML.
* <script src="myScript.js"></script>
  + To use external js
  + External scripts cannot contain <script> tags.
  + We can use a full URL also in src
* Output
  + Writing into an HTML element, using innerHTML.
  + Writing into the HTML output using document.write().
  + Writing into an alert box, using window.alert().
  + Writing into the browser console, using console.log().
    - For debugging purposes using F12 in debugger menu’s console
  + Using document.write() after an HTML document is loaded, will **delete all existing HTML**
    - should only be used for testing.
  + In HTML, JavaScript programs are executed by the web browser
* Syntax:
  + Hyphens are not allowed in JavaScript. They are reserved for subtractions
* Operators:
  + A variable declared without a value will have the value undefined.
  + If you put a number in quotes, the rest of the numbers will be treated as strings, and concatenated.
  + === equal type and equal value
  + typeOf -> returns the type of variable
  + instanceOf -> returns true if obj is an instance of an obj type
  + Arithmetic operators:
    - +(addition), -(subtraction), \*(multiplication), /(division), %(modulus), ++(increment), --(decrement)
  + Comparison operator:
    - ==, !=, >, <, >=, <=
  + Logical operators:
    - && -> logical AND
    - || -> logical OR
    - ! -> logical NOT
  + bitwise operator
    - & - AND, | - OR, ~ - NOT, ^ - XOR
    - << - zero fill left shift
    - >> - signed right shift
    - >>> - zero fill right shift
    - The examples above uses 4 bits unsigned examples. But JavaScript uses 32-bit signed numbers.  
      Because of this, in JavaScript, ~ 5 will not return 10. It will return -6.  
      ~00000000000000000000000000000101 will return 11111111111111111111111111111010
  + Assignment operators:
    - =, +=, -=, \*=, /=, %=
    - Same logic applies to Bitwise operators so they will become like <<=, >>=, >>=, &=, |= and ^=.
  + Conditional operator –
    - **? : (Conditional )**
    - If Condition is true? Then value X : Otherwise value Y
  + typeof operator
    - typeof(Null) - Object
* Data Types:
  + - JavaScript has dynamic types.
      * means that the same variable can be used to hold different data types
      * means you don't need to specify type of the variable
      * var x;           // Now x is undefined  
         x = 5;           // Now x is a Number  
         x = "John";      // Now x is a String
    - Primitive Data Types:
      * It is called primitive bcoz, their values can contain only a single thing (be it a string or a number or whatever).
      * String, number, Boolean, undefined, null
    - Non-primitive data types:
      * Object, array, regExp
    - You can use quotes inside a string, as long as they don't match the quotes surrounding the string
      * var answer2 = "He is called 'Johnny'";    // Single quotes inside double quotes  
         var answer3 = 'He is called "Johnny"';    // Double quotes inside single quotes
    - Any variable can be emptied, by setting the value to undefined. The type will also be undefined
      * car = undefined;    // Value is undefined, type is undefined
    - In JavaScript null is "nothing".
      * Unfortunately, in JavaScript, the data type of null is an object.
      * You can consider it a bug in JavaScript that typeof null is an object. It should be null.
      * You can empty an object by setting it to null:
        + var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};  
          person = null;    // Now value is null, but type is still an object
      * You can also empty an object by setting it to undefined
    - typeof undefined           // undefined  
       typeof null                // object
    - typeof Array is Object
    - undefined and null are equal in value but different in type
    - Primitive data:
      * The typeof operator can return one of these primitive types:
        + String, number, Boolean, undefined
    - Complex data:
      * The typeof operator can return one of two complex types:
        + Object - for objects, arrays, and null.
        + Function - The typeof operator does not return "object" for functions.
        + Note: The typeof operator returns "object" for arrays because in JavaScript arrays are objects.
* Functions:
  + Function **parameters** are listed inside the parentheses () in the function definition.
  + Function **arguments** are the **values** received by the function when it is invoked.
  + Accessing a function without () will return the function definition instead of the function result: function toCelsius(f) { return (5/9) \* (f-32); }
  + Local variables are created when a function starts, and deleted when the function is completed.
* Objects: has property & methods
  + JavaScript objects are containers for **named values**called properties or methods.
  + Accessing obj properties:
    - objectName.propertyName
    - objectName[“propertyName”]
  + A method is a function stored as a property.
  + Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.
    - var x = new String();        // Declares x as a String object  
       var y = new Number();        // Declares y as a Number object  
       var z = new Boolean();       // Declares z as a Boolean object
* Events:
  + An HTML event can be something the browser does, or something a user does.
    - An HTML web page has finished loading
    - An HTML input field was changed
    - An HTML button was clicked
    - <element event=**'*some JavaScript*'**>
    - For eg:
      * <button onclick="document.getElementById('demo').innerHTML = Date()">The time is?</button>
* Strings:
  + Txt.length – string length
  + **backslash escape character** 
    - **\” -> “**
    - **\’ -> ‘**
    - A safer way to break up a string, is to use string addition(concat)
* String methods:
  + Primitive values, like "John Doe", cannot have properties or methods (because they are not objects).
    - * + But with JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.
  + Both indexOf(), and lastIndexOf() return -1 if the text is not found.
  + search() :  method searches a string for a specified value and returns the position of the match
  + The two methods, indexOf() and search(), are **equal?**
    - * search() method cannot take a second start position argument.
      * indexOf() method cannot take powerful search values (regular expressions).
  + substring()
    - is similar to slice().
    - The difference is that substring() cannot accept negative indexes.
  + substr()
    - is similar to slice().
    - The difference is that the second parameter specifies the **length** of the extracted part.
  + replace() method replaces a specified value with another value in a string
    - it does not change the string it is called on. It returns a new string.
    - replaces **only the first** match
    - /g – global
    - /i – case-insensitive
  + toUpperCase()
  + toLowerCase()
  + concat()
    - can be used instead of the plus operator
  + NOTE: All string methods return a new string. They don't modify the original string.  
    Formally said: Strings are immutable: Strings cannot be changed, only replaced.
  + The trim() method removes whitespace from both sides of a string
  + Extracting String Characters:
    - charAt(*position*)
    - charCodeAt(*position*)
    - Property access [ ]
    - var str = "HELLO WORLD";  
       str[0] = "A";             **// Gives no error, but does not work**  
       str[0];                   // returns H
* Numbers:
  + This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63
* Arrays:
  + JavaScript arrays are used to store multiple values in a single variable
  + the full array can be accessed by referring to the array name
  + typeof(array) – Object
  + You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array
    - myArray[0] = Date.now;  
      myArray[1] = myFunction;  
      myArray[2] = myCars;
  + In JavaScript, **arrays** use **numbered indexes**.
    - use **arrays** when you want the element names to be **numbers**
  + In JavaScript, **objects** use **named indexes**.
    - use **objects** when you want the element names to be **strings (text)**
  + Note: Arrays are a special kind of objects, with numbered indexes.
  + Array.isArray(fruits);   // returns true <- ES6
    - To solve the problem, that typeof operator returns object for both array and object, i.e to find if a variable is an array?
  + JavaScript automatically converts an array to a comma separated string when a primitive value is expected
    - This is always the case when you try to output an array
    - All JavaScript objects have a toString() method
  + Even if objects have properties of different data types, the sort() method can be used to sort the array.
    - The solution is to write a compare function to compare the property values
* Scope:
  + Two types:
    - Local scope-
      * Local variables are created when a function starts, and deleted when the function is completed
    - Global scope-
  + Scope determines the accessibility of variables, objects, and functions from different parts of the code.
  + Strict mode-
    - All modern browsers support running JavaScript in "Strict Mode"
    - In "Strict Mode", undeclared variables are not automatically global.
  + Warning-
    - Your global variables (or functions) can overwrite window variables (or functions).  
      Any function, including the window object, can overwrite your global variables and functions.
  + Lifetime of global variables-
    - Local – deleted when function gets completed.
    - In web browser, Global – gets deleted when we close the browser
  + Function Arguments-
    - Function arguments (parameters) work as local variables inside functions.
* Iterators and Generators:  
  + Iterators-
    - It is an object, acts as an interface to the object (Array, string, map, set, NodeList – these have built-in iterators)
    - It’s an object, that gives back contents from some other object, in some pre-determined order.
    - Used to step through the properties and return them
  + Generators:
    - These are functions
    - Put asterick after function keyword to make it a generator
    - We can configure the order by which we’re getting things back
    - Generators are a special class of functions that simplify the task of writing iterators
    - A generator is a function that produces a sequence of results instead of a single value, i.e you generate ​a series of values
    - In JavaScript, a generator is a function which returns an object on which you can call next(). Every invocation of next() will return an object of shape —  
        
      * {   
         value: Any,  
         done: true|false  
        }
  + [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String), [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), [TypedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray" \o "A TypedArray object describes an array-like view of an underlying binary data buffer. There is no global property named TypedArray, nor is there a directly visible TypedArray constructor.  Instead, there are a number of different global properties, whose values are typed array constructors for specific element types, listed below. On the following pages you will find common properties and methods that can be used with any typed array containing elements of any type.), [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) and [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set) are all built-in iterables, because their prototype objects all have a [Symbol.iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator" \o "The Symbol.iterator well-known symbol specifies the default iterator for an object. Used by for...of.) method
  + <https://codeburst.io/understanding-generators-in-es6-javascript-with-examples-6728834016d5>
* Prototypes:
  + JavaScript is often described as a **prototype-based language**
  + provide inheritance, objects can have a **prototype object**, which acts as a template object that it inherits methods and properties from
  + The Object.prototype is on the top of the prototype inheritance chain
    - Date objects, Array objects, and Person objects inherit from Object.prototype.
  + The JavaScript prototype property allows you to add new properties to object constructors
  + **When a certain method(or property) is called, it first checks inside the object but when it doesn’t find, then search moves on Object’s prototype**
  + **Prototype property enables other objects to inherit all the properties and methods of function constructor.**
* **Recursion:**
  + **2 main parts:**
    - **Terminating condition**
    - **Recursive case**
  + **The recursive func calls are made, then all the calls gets stacked up in the stack, now, if the no. of func calls are more than the Browser limit (every browser have diff. limit), then the browser will throw a StackOverFlow error.**
  + **This problem can be solved by using , Tail call optimization (ES6 features), but as of now it hasn’t been implemented in almost all of the browsers.**
  + **This algorithm is written just like normal recursion( but in the recursion case, don’t use any local variables), but works only in some browsers upto now. This algo doesn’t stacks up the calls, instead frees up stacks after each call.**
  + Note: You have to use strict mode to have access to tail call optimization
* Hoisting:
  + a variable can be used before it has been declared
  + Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).
    - Variables and constants declared with let or const are not hoisted!
  + But, it doesn’t work with initializations, i.e
    - var a =5 // initialize a
  + JavaScript in strict mode does not allow variables to be used if they are not declared.
  + Here, we can also make a call to the function before defining it.
* Try-catch :
  + Steps-
    - First, the code in try {...} is executed.
    - If there were no errors, then catch(err) is ignored: the execution reaches the end of try and goes on, skipping catch.
    - If an error occurs, then the try execution is stopped, and control flows to the beginning of catch(err). The err variable (we can use any name for it) will contain an error object with details about what happened.
  + **try..catch only works for runtime errors**
    - For try..catch to work, the code must be runnable. In other words, it should be valid JavaScript.
* Closure:
  + In any function, if you’re using a variable from outside the scope, are closures.
* Event Loop:
  + It is a data structure which records basically where in a program we are, so if we step into a function, we put something onto the stack, if we return from a function, we pop off the top of the stack.
  + The chrome browser gives the call stack trace
  + Blowing the stack means: results in RangeError: Max call stack size exceeded
  + If we run any code in JS, if it is taking n/w requests, then the browser will wait for the completion of whole program , and the Browser gets into a blocking state, i.e it can’t do anything until all the requests gets completed. So, here the browser can’t run any other code, it’s stuck.
* ES5
  + ES5 New Object Methods
    - // Adding or changing an object property  
      Object.defineProperty(object, property, descriptor)  
        
      // Adding or changing many object properties  
      Object.defineProperties(object, descriptors)  
        
      // Accessing Properties  
      Object.getOwnPropertyDescriptor(object, property)  
        
      // Returns all properties as an array  
      Object.getOwnPropertyNames(object)  
        
      // Returns enumerable properties as an array  
      Object.keys(object)  
        
      // Accessing the prototype  
      Object.getPrototypeOf(object)  
        
      // Prevents adding properties to an object  
      Object.preventExtensions(object)  
      // Returns true if properties can be added to an object  
      Object.isExtensible(object)  
        
      // Prevents changes of object properties (not values)  
      Object.seal(object)  
      // Returns true if object is sealed  
      Object.isSealed(object)  
        
      // Prevents any changes to an object  
      Object.freeze(object)  
      // Returns true if object is frozen  
      Object.isFrozen(object)
* ES6
  + Arrow Function
    - Arrow functions do not have their own this. They are not well suited for defining **object methods**.
    - Arrow functions are not hoisted. They must be defined **before** they are used.
    - Using const is safer than using var, because a function expression is always constant value.
    - You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them