document.getElementById("demo")**.innerHTML** = 5 + 6;

**document.write**(5 + 6); useful in testing, **delete all existing HTML, if we call post html load**

**console.log**(5 + 6); debugging, display on browser console

window.**alert**(5 + 6); popup

* JS statements are executed, one by one, in the same order as they are written
* Ending statements with semicolon is not required, but highly recommended.
* If a JS statement does not fit on one line, the best place to break it, is after an operator
* JS has two type value, Fixed values are called **literals**. Variable values are called **variables**.
* Variable name is Identifiers , Eg var munnaVal = 10; munnaVal is identifiers, **case sensitive**.
* equal sign (=) is an "assignment" operator
* A variable declared without a value will have the value **undefined**.
* If you re-declare a JavaScript variable, it will not lose its value. (untill assign again)
* same variable can be used to hold different data types( var x= 7, var x=true, var x=”string”)
* **You can also empty an object by setting it to undefined**
* Undefined and null are equal in value but different in type null === undefined -> false, null == undefined -> true
* Accessing a function without () will return the function definition instead of the function result, You can use the function directly, as a variable value

2 + 1 \* 3 \* 1 - 2 / 2; 🡪 MULT FIRST, DIVISION SECOND, + AND THEN -

* **Code Blocks** -> statements can be grouped together in code blocks, inside curly brackets {..} / methods
* **keyword** -> break, continue, debugger, do, while, for, function, if, return, switch, try, var
* **types** -> string, number, Obj, Array, udefined, null, function, boolean (**typeof(**var**)** used to get types)
* **Scope** -> determines the **accessibility (visibility) of these variables**. Local variables have **local scope**: They can only be accessed within the function. A variable declared outside a function, becomes **GLOBAL**. All scripts and functions on a web page can access it.
* Automatically Global -> If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.
* Local variables are deleted when the function is completed.
* global variables are deleted when you close the browser window (or tab), but remain available to new pages loaded into the same window.
* the global scope is the window object. All global variables belong to the window object.

All **global** JavaScript objects, functions, and variables automatically become members of the **window object**. Even the document object (of the HTML DOM) is a property of the window object:

window.document.getElementById("header");

* window.innerHeight - the inner height of the browser window (in pixels)
* window.innerWidth - the inner width of the browser window (in pixels)
* window.open() - open a new window
* window.close() - close the current window
* window.moveTo() -move the current window
* window.resizeTo() -resize the current window

The **window.screen** object can be written without the window prefix.

* screen.width
* screen.height
* screen.availWidth
* screen.availHeight
* screen.colorDepth
* screen.pixelDepth
* **window.location.href** returns the href (URL) of the current page
* **window.location.hostname** returns the domain name of the web host
* **window.location.pathname** returns the path and filename of the current page
* **window.location.protocol** returns the web protocol used (http: or https:)
* **window.location.assign** loads a new document
* **window.location.port**
* **history.back() - same as clicking back in the browser**
* **history.forward() - same as clicking forward in the browser**

document.getElementById("demo").innerHTML = navigator.cookieEnabled; //returns true if cookies are enabled

navigator.appName; // returns the application name of the browser (netscope)

navigator.appCodeName; //  returns the application code name of the browser (mozila, chrome etc,.)

JavaScript has three kind of popup boxes: Alert box, Confirm box, and Prompt box.

* setTimeout(*function, milliseconds*)  
  Executes a function, after waiting a specified number of milliseconds.
* setInterval(*function, milliseconds*)  
  Same as setTimeout(), but repeats the execution of the function continuously.

The setTimeout() and setInterval() are both methods of the HTML DOM Window object.

var myVar = setInterval(myTimer, 1000);  
function myTimer() {  
    var d = new Date();  
    document.getElementById("demo").innerHTML = d.toLocaleTimeString();  
}

clearInterval(myVar) // to clear timeinterval

Events **->** Mouse Events **(onclick, ondblclick, onmousedown, onmouseenter, onmouseleave, onmouseover)**

KeyboardEvents **(Onkeyup, OnKeydown, Onkeypress)**

ClipboardEvents **(oncopy, oncut, onpaste)**

Print Events **(onafterprint, onbeforeprint)**

Form event **(onblur, onchange, onfocus, onsubmit etc.,)**

Drag events **(ondrop, ondrag etc.,) etc.,**

The **prototype** property allows you to add new properties and methods to existing object types.

String - > properities (Constructor -> function String(){ [native] } , Prototype, length)

**Methods (indexOf, lastIndexOf, Includes, CharAt, trim, Slice, Split, Concat, toString, toLowerCase, toUpperCase, substr, startwith, endwith)**

* Var str = “Munna”; Var str2 = “Babu”
* str.**charAt**(4); // a
* str.**concat**(str2); // Munna Babu
* str.**endsWith**("na"); // True , fname.endsWith("NA"); // False
* str.**trim**() 🡪 remove whitespace from both side of string
* str.**includes**("world") // check str include of world, return true , false. Case sensitive
* str.**indexOf**("Mu") // will return index of string, not found -1. first occurance index
* str.**lastIndexOf**("Mu") // will return index of string, not found -1.last occurance index
* str.**replace**("Microsoft", "W3Schools");
* str.**split**(" "); // Spilit str to array
* str.**search**("locate"); // same like indexOf, return position of str. INTEGER
* var y = new String("John"); // console.log(y) – John
* str.**substr**(1, 4); (start, length) // unna , Extract part of string same like slice but it will return end index
* str.**slice**(1, 4); // (start, end). Output -> unn,
* **slice**(start, end) , use –negative for last to front
* **substring**(start, end) same slice, can’t use -ve
* **substr**(start, length)

NaN is a JavaScript reserved word indicating that a number is not a legal number.

Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number):

**parseInt()** parses a string and returns a whole number. Spaces are allowed. Only the first number is returned

parseInt("10");         // returns 10  
parseInt("10.33");      // returns 10  
parseInt("10 20 30");   // returns 10  
parseInt("10 years");   // returns 10  
parseInt("years 10");   // returns NaN

parseFloat("10");        // returns 10  
parseFloat("10.33");     // returns 10.33  
parseFloat("10 20 30");  // returns 10  
parseFloat("10 years");  // returns 10  
parseFloat("years 10");  // returns NaN

Number(true);          // returns 1  
Number(false);         // returns 0  
Number("10");          // returns 10  
Number("  10");        // returns 10  
Number("10  ");        // returns 10  
Number("10 20");       // returns NaN   
Number("John");        // returns NaN

Math.round(4.7);    // returns 5  
Math.round(4.4);    // returns 4

Math.pow(8, 2);      // returns 64 Math.pow(x, y) returns the value of x to the power of y

Math.sqrt(64);      // returns 8 Math.sqrt(x) returns the square root of x

Math.min(0, 150, 30, 20, -8, -200);  // returns -200

Math.max(0, 150, 30, 20, -8, -200);  // returns 150

Math.random();     // returns a random number

Math.floor(Math.random() \* 100);     // returns a number between 0 and 99

Math.floor(Math.random() \* 10) + 1;  // returns a number between 1 and 10

Math.floor(Math.random() \* (max - min) + 1 ) + min;

Array.isArray(fruits);     // returns true

fruits instanceof Array     // returns true

**ARRAY:**

**Push, pop, shift, unsshift, splice, slice, concat, join, filter, every, foreach, map**

The **push()** method adds a new element to an array (at the end):

var fruits = ["Banana", "Orange", "Apple", "Mango"];

var x = fruits.**push**("Kiwi");   //  Adds a new element ("Kiwi") to fruits, the value of x is 5 return new array length

The **pop()** method removes the last element from an array at end

var x = fruits.**pop**();         // Removes the last element ("Mango") from fruits, x= mango

fruits.**toString()** converts an array to a string

fruits.**join**(" , "); join array and return as string

Shifting is equivalent to popping, working on the first element instead of the last.

fruits.**shift**();    // Removes the first element "Banana" from fruits, return mango

The **unshift()** method adds a new element to an array (at the beginning), and "unshifts" older elements:

fruits.**unshift**("Lemon");    // Adds a new element "Lemon" to fruits , // Returns 5 new array length

Slice same like in string

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];

var citrus = fruits.slice(1, 3);

*array*.splice(index, howmanyRemove, item1, ....., itemX)

The **splice()** method can be used to add new items to an array

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.**splice**(2, 0, "Lemon", "Kiwi");

The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

The second parameter (0) defines **how many** elements should be **removed**.

The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

fruits.**splice**(0, 1);        // Removes the first element of fruits

The first parameter (0) defines the position where new elements should be **added** (spliced in).

The second parameter (1) defines **how many** elements should be **removed**.

The rest of the parameters are omitted. No new elements will be added.

var myChildren = myGirls.concat(myBoys);     // Concatenates (joins) myGirls and myBoys

var myChildren = arr1.concat(arr2, arr3);     // Concatenates arr1 with arr2 and arr3

The slice() method creates a new array. It does not remove any elements from the source array.

ARRAY METHODS - > slice, splice, shift, pop, unshift, push, indexOf, Includes, findIndexOf, lastIndexOf, Join, Concat, toString, Filter, Map, forEach, Every

**Note:** filter() does not change the original array.

The findIndex() method returns the index of the first element in an array that pass a test (provided as a function).

array.forEach(function(currentValue, index, arr), thisValue)

points.sort(function(a, b){return a-b});    // Sort the numbers in the array in ascending order

const selectAll = chkAll.every((item: boolean) => item === true);

this.dropdownValue = this.dropdownValue.filter(obj => obj.id !== selection.id);

this.dropdownValue = this.dropdownValue.findIndexOf(obj => obj.id !== selection.id);

function checkAdult(age) {  
    return age >= 18;  
}

ages.every(checkAdult)

The every() method checks if all elements in an array pass a test (provided as a function).

var ages = [32, 33, 16, 40];  
  
ages.filter(checkAdult)

function myFunction() {  
    document.getElementById("demo").innerHTML = ages.every(checkAdult);  
}

switch(expression) {  
    case n:  
        code block        break;  
    case n:  
        code block        break;  
    default:  
        code block  
}

for (x in person) {  
    text += person[x];  
}

The break statement "jumps out" of a loop.

The continue statement "jumps over" one iteration in the loop.

## JavaScript Data Types

In JavaScript there are 5 different data types that can contain values:

* string
* number
* boolean
* object
* function

There are 3 types of objects:

* Object
* Date
* Array

And 2 data types that cannot contain values:

* null
* undefined

try {  
    x = y + 1;   // y cannot be referenced (used)  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.name;  
}

 var x = document.forms["myForm"]["fname"].value;

Parse the data with **JSON.parse(),** and the data becomes a JavaScript object.

Convert a JavaScript object into a string with **JSON.stringify()**.

* display:none
* Remove
* visibility:hidden
* hide

(function () {  
    var x = "Hello!!";      // I will invoke myself  
})();

The arguments.length property returns the number of arguments received when the function was invoked

function myFunction(a, b) {  
    return arguments.length;  
}

If a function is called with **missing arguments** (less than declared), the missing values are set to: **undefined**

// 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;                             // Will return "John"

With **call()**, you can use a method belonging to another object.

var person = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
var myObject = {  
    firstName:"Mary",  
    lastName: "Doe",  
}  
person.fullName.call(myObject);  // Will return "Mary Doe"

var add = (function () {  
    var counter = 0;  
    return function () {return counter += 1;}  
})();  
  
add();  
add();  
add();  
  
// the counter is now 3

The variable **add** is assigned the return value of a self-invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

**JavaScript prototype** property allows you to add new properties to object constructors

function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}

Person.prototype.name = function() {  
    return this.firstName + " " + this.lastName;  
};

var myFather = new Person("John", "Doe", 50, "blue"); // Contructor function

**Abstract Class** -- > Inheritance Base(super class), If we mention Abstract Class ClassName { }, We can’t directly access base(Abstract class properties , methods, can’t create object to abstract class). But we can create obj to derived class thru that can access abstract class

**Abstract methods: 🡪**  If we declare function with abstract key in base class, then derived class must have that functions in derived classes.

**If base class has constructor, then derived must call base class constructor from derived class constuctor using super()**

**Class closure**

function greet(message) {

console.log(message);

}

function greeter(name, age) {

return name + " says howdy!! He is " + age + " years old";

}

var message = greeter("James", 23);

greet(message);

function greeter(name, age) {

var message = name + " says howdy!! He is " + age + " years old";

return function greet() {

console.log(message);

};

}

var JamesGreeter = greeter("James", 23);

JamesGreeter();

The closure has access to variables in three scopes; specifically: (1) variable in its own scope, (2) variables in the enclosing function’s scope, and (3) global variables.

**Function factories**: One powerful use of closures is to use the outer function as a factory for creating functions that are somehow related.

function dwightJob(title) {  
 return function(prefix) {  
 return prefix + ' ' + title;  
 };  
}  
var sales = dwightJob('Salesman');  
var manager = dwightJob('Manager');

alert(manager('Regional')); // Regional Manager  
alert(sales('Top')); // Top Salesman

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

var dwightSalary = (function() {  
 var salary = 60000;  
 function changeBy(amount) {  
 salary += amount;  
 }  
 return {  
 raise: function() {  
 changeBy(5000);  
 },  
 lower: function() {  
 changeBy(-5000);  
 },  
 currentAmount: function() {  
 return salary;  
 }  
 };   
})();  
  
alert(dwightSalary.currentAmount()); // $60,000  
dwightSalary.raise();  
alert(dwightSalary.currentAmount()); // $65,000  
dwightSalary.lower();  
dwightSalary.lower();  
alert(dwightSalary.currentAmount()); // $55,000  
  
dwightSalary.changeBy(10000) // TypeError: undefined is not a function

a = (function () {

var privatefunction = function () {

alert('hello');

}

return {

publicfunction : function () {

privatefunction();

}

}

})();

function countWrapper() {

var counter = 0;

function updateClickCount() {

++counter;

// do something with counter

}

updateClickCount();

return counter;

}

var updateClickCount=(function(){

var counter=0;

return function(){

++counter;

// do something with counter

}

})();

document.**getElementById**(id) Find an element by element id

document.**getElementsByTagName**(name) Find elements by tag name

document.**getElementsByClassName**(name) Find elements by class name

element.**setAttribute**(attribute, value) Change the attribute value of an HTML element

element.**style.property** = new style Change the style of an HTML element

element.**innerHTML** = new html content Change the inner HTML of an element

element.**attribute** = new value Change the attribute value of an HTML element

document.**createElement**(element) Create an HTML element

document.**removeChild**(element) Remove an HTML element

document.**appendChild**(element) Add an HTML element

document.**replaceChild**(element) Replace an HTML element

document.**write**(text) Write into the HTML output stream

document.getElementById(id).**onclick** = function(){code}

document.**baseURI** Returns the absolute base URI of the document

document.**body**  Returns the <body> element

document.**documentURI** Returns the URI of the document

document.**domain** Returns the domain name of the document server

document.**forms** Returns all <form> elements

document.**head** Returns the <head> element

document.**images** Returns all <img> elements

document.**title** Returns the <title> element

document.**URL** Returns the complete URL of the document

document.getElementById("myBtn").**addEventListener("click", displayDate);**

element.**addEventListener**("click", mySecondFunction);

box-modelling 🡪 content, padding, border, margin. Box-sizing: border-box🡪 border will be part of content

specifity 🡪 if two style for same div will pick second one, if we provide div with class div.classname. then div.class wil be taken, if div#id and div.class then id will be taken. If !important given important will be taken

relative 🡪 it will not disturb other element, we can position top, left, botom

absolute -> will disturb next element, next element will take current element space

**SORT ARRAY without any methods:**

public sortArray(array) {

for (const ar of array) {

for (let i = 0; i < array.length; i++) {

if (array[i] > array[i + 1]) { // ASCENDING, < descending

const tmp = array[i + 1];

array[i + 1] = array[i];

array[i] = tmp;

}

}

}

return array;

}

return array.sort((a, b) => a-b; ); // ASCENDING

fruits.sort(); // for string work ["Banana", "Orange", "Apple", "Mango"]

var points = [40, 100, 1, 5, 25, 10];  
points.sort((a, b) => {return a-b});

Merge two array (Without concat, join)

const x = [1, 2, 3];

const y = [4, 5, 6];

x.push.apply(x, y); // Method 1

a = [...x, ...y]; // Method 2

x.push(...y); // Method 3

x 🡺 [1,2,3,4,5,6]

**FIND EACH character count on string:**

const tmp = {};

const temstr = 'Munna'.split('');

for (const char of temstr) {

tmp[char] = tmp[char] ? tmp[char] + 1 : 1;

}

**Difference of two Array:**

const a1 = ['1', '2', '3', '4', '6'];

const a2 = ['3', '4', '5'];

a1.concat(a2).filter((val, index, arr) => arr.indexOf(val) === arr.lastIndexOf(val)) // 1,2,5,6

a1.filter( (val) => a2.indexOf(val) !== -1 ) // 3,4

a1.filter( (val) => a2.indexOf(val) === -1 ) // 1,2,6

**Prime Number:**

function getPrimes(max) {

var tmp = [];

var i;

var j;

var primes = [];

for (i = 2; i <= max; ++i) {

if (!tmp[i]) {

primes.push(i);

for (j = i << 1; j <= max; j += i) {

tmp[j] = true;

}

}

}

return primes;

}

console.log(getPrimes(100));

**factorial Number:**

function show(num){

var fact=1;

for(let i=1; i <= num; i++) {

fact= fact\*i;

}

return fact;

}

function isEquivalent(a, b) {

var aProps = Object.getOwnPropertyNames(a);

var bProps = Object.getOwnPropertyNames(b);

if (aProps.length != bProps.length) {

return false;

}

for (var i = 0; i < aProps.length; i++) {

var propName = aProps[i];

if (a[propName] !== b[propName]) {

return false;

}

}

return true;

}

console.log(isEquivalent(bobaFett, jangoFett));

'999': /^([0-9]){0,3}$/g,

'9999': /^([0-9]){0,4}$/g,

'preventDefault': /^([0-9-]){0,7}$/g,

'decimalTwo': /^\d{0,6}\.?\d{0,2}$/g,

'9999.999': /^\d{0,7}\.?\d{0,3}$/g,

'alpha25': /^([a-zA-Z0-9]){0,25}$/g,

'number8Wild': /^([0-9\*]){0,8}$/g,

'alpha10Wild': /^([a-zA-Z0-9\*]){0,10}$/g,

'alpha9': /^([a-zA-Z0-9]){0,9}$/g,

'time': /^([0-9]){0,2}$/g,

'dateFormat': /^\d{0,2}\/?\d{0,2}\/?\d{0,4}$/g,

'timeHour': /^([0-1]?[0-9]|2[0-3])$/g,

'timeMin': /^([0-5]?[0-9])$/g

myValue.match(myExpression);

**/^ () {} $/g**

export class DxcDisableService {

public bookingClasses = new Subject <string> ();

public setBooking(currentValue: string) {

this.bookingClasses.next(currentValue);

}

public getBooking(): Observable <string> {

return this.bookingClasses.asObservable();

}

}

# **Rest Parameters**

**function** **buildName**(firstName: string, ...restOfName: string[]) {

**return** firstName + " " + restOfName.join(" ");

}

**let** employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");

**let** myAdd: (x: number, y: number) => number = **function**(x: number, y: number): **number** { **return** x + y; };

**ViewContainerRef** - Represents a container where one or more Views can be attached.

Validations:

this.heroForm = new FormGroup({

'name': new FormControl(this.hero.name, [

Validators.required,

Validators.minLength(4),

this.forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.

])

})

forbiddenNameValidator(nameRe: RegExp): ValidatorFn {

return (control: AbstractControl) => {

const forbidden = nameRe.test(control.value);

return forbidden ? {'forbiddenName': {value: control.value}} : null;

};

Authentications : ROUTE canActivate

AUTH.GURARD.ts

export class AuthGuard implements canActivate {

Constructor(home: homeService, router: Router) {

}

canActivate( next: ActivatedRouteSnapshot, state: RouterStateSnapshot ): boolean {

if (this.home.checkLoggin()) {

return true;

}else {

this.router.navigate(['/loggin']);

return false;

}

}

}

Routes= [{path: ‘home’, canActivate: ‘AuthGuard’, component: ComponentClassName}]

Dynamic components:

import { ClassesComponent } from '../conditions/classes/classes.component';

@ViewChild('classesContainer', {read: ViewContainerRef})

public classesContainer: ViewContainerRef;

const resolver = this.factoryResolver.resolveComponentFactory(ClassesComponent);

const classesContRef = this.classesContainer.createComponent(resolver);

classesContRef.instance.childInput = this.ruleViewData;

HTTP CACHE:

***METHOD 1:***

getFriends(){

if(!this.\_friends) {

this.\_friends = this.\_http.get('./components/rxjs-caching/friends.json')

.map((res:Response) => res.json().friends).**publishReplay(1).refCount();**

}

return this.\_friends;

}

publishReplay(1) tells rxjs to cache the most recent value which is perfect for single value http calls. refCount() is used to keep the observable alive for as long as there are subscribers.

***METHOD 2:***

// service.ts

$observable;

getUserDetail(){

if(!this.observable$){

this.observable$ = this.http.get(this.\_userDetailsURL)

.map(res => res.json())**.shareReplay();**

}

return this.observable$.toPromise();

}

clearUserCache(){

this.observable$ = null;

}

***METHOD 3:***

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/observable/of'; //proper way to import the 'of' operator

import 'rxjs/add/operator/share';

import 'rxjs/add/operator/map';

@Injectable()

export class DataService {

private url: string = 'https://cors-test.appspot.com/test';

private data: any;

private observable: Observable<any>;

constructor(private http: Http) { }

getData() {

if (this.data) {

return Observable.of(this.data);

} else if (this.observable) {

return this.observable;

} else {

this.observable = this.http.get(this.url).map(response => {

this.observable = null;

this.data = response.json();

return this.data;

}).share(); // make it shared so more than one subscriber can get the result

return this.observable;

}

}

}

Redux -

action --> type, payload

state --> previous value

this.store.dispatch({type: 'myType', payload: 'myData'}) --> store value(it call reducer function and store the change to post(singleObj));

this.store.select(‘post’);