**Typescript Tutorial**

**Lesson 01 Introduction to Typescript tutorial**

**Notes:-**

**1-ES5 = ECMA Script 5 (is supported by all the browsers)**

**2-JavaScript is the one of the implementation of the standard called Ecma Script**

**3-ECMA Script 6 (ECMA 2015) is the new version = ECMA 5 + new features**

**Typescript features:-**

**1-Open Source**

**2-Microsoft built in**

**3-is superset of JavaScript**

**4-Compiles to plain JavaScript**

**5-Features of ES2015 & 2016**

**6-Types (Optional) 🡺 means number , string ,……,enum**

**7-Safer Code**

**Lesson02 Transpiler**

**Notes:-**

**1-browser like chrome, firefox implement almost 90% of the Ecma 2015 the support type Script**

**2-The transpiler is the compiler that compiles / transfers the TypeScript code to JavaScript code**

**Lesson03 Tooling Setup**

**We will use the following tools**

**1-Visual Studio Code**

**2- Npm (node package manager):- is the manager that manage the dependency and execution of the Typescript**

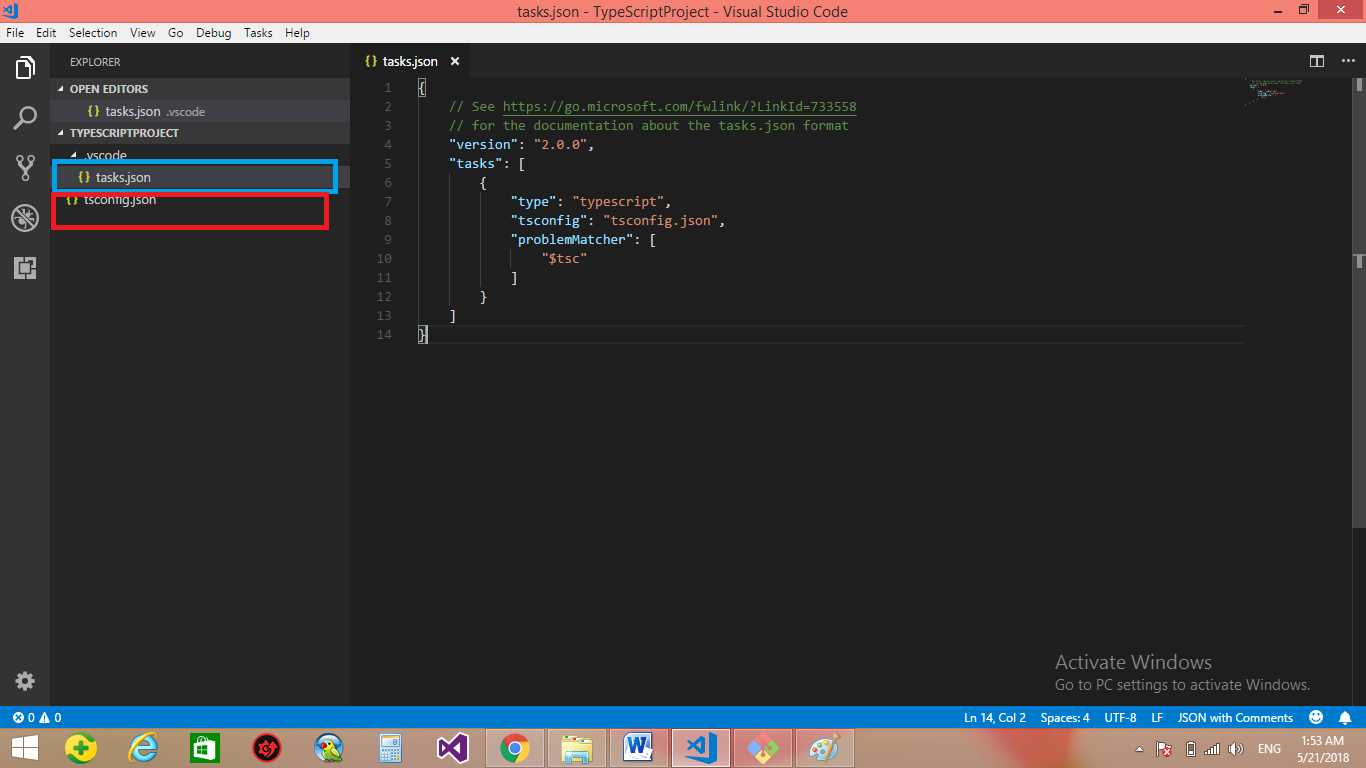
**3-Node.js (one you install node.js, the NPM will automatically installed)**

**4-after install node.js on the command prompt we will type**

**Npm install typescript –g**

**//this will install the latest version of the typescript compiler**

**5-we will create folder called TypeScriptFolder in desktop and then open that folder on visual studio code and then make ctrl+Shift+B and then select manage configure and then select task.json**

****

**6-on the folder that we will open we will open the visual studio code on it ,and then type the following code on cmd**

**Tsc –init //this will create tsconfig.json file**

**{**

**"compilerOptions": {**

**/\* Basic Options \*/**

**"target": "es5", /\* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017','ES2018' or 'ESNEXT'. \*/**

**"module": "commonjs", /\* Specify module code generation: 'none', 'commonjs', 'amd', 'system', 'umd', 'es2015', or 'ESNext'. \*/**

**// "lib": [], /\* Specify library files to be included in the compilation. \*/**

**// "allowJs": true, /\* Allow javascript files to be compiled. \*/**

**// "checkJs": true, /\* Report errors in .js files. \*/**

**// "jsx": "preserve", /\* Specify JSX code generation: 'preserve', 'react-native', or 'react'. \*/**

**// "declaration": true, /\* Generates corresponding '.d.ts' file. \*/**

**"sourceMap": true, /\* Generates corresponding '.map' file. \*/**

**// "outFile": "./", /\* Concatenate and emit output to single file. \*/**

**//to redirect the output to specific path we declare**

**"outDir": "scripts/", /\* Redirect output structure to the directory. \*/**

**// "rootDir": "./", /\* Specify the root directory of input files. Use to control the output directory structure with --outDir. \*/**

**// "removeComments": true, /\* Do not emit comments to output. \*/**

**// "noEmit": true, /\* Do not emit outputs. \*/**

**// "importHelpers": true, /\* Import emit helpers from 'tslib'. \*/**

**// "downlevelIteration": true, /\* Provide full support for iterables in 'for-of', spread, and destructuring when targeting 'ES5' or 'ES3'. \*/**

**// "isolatedModules": true, /\* Transpile each file as a separate module (similar to 'ts.transpileModule').**

**/\* Strict Type-Checking Options \*/**

**"strict": true, /\* Enable all strict type-checking options. \*/**

**// "noImplicitAny": true, /\* Raise error on expressions and declarations with an implied 'any' type. \*/**

**// "strictNullChecks": true, /\* Enable strict null checks. \*/**

**// "strictFunctionTypes": true, /\* Enable strict checking of function types. \*/**

**// "strictPropertyInitialization": true, /\* Enable strict checking of property initialization in classes. \*/**

**// "noImplicitThis": true, /\* Raise error on 'this' expressions with an implied 'any' type. \*/**

**// "alwaysStrict": true, /\* Parse in strict mode and emit "use strict" for each source file. \*/**

**/\* Additional Checks \*/**

**// "noUnusedLocals": true, /\* Report errors on unused locals. \*/**

**// "noUnusedParameters": true, /\* Report errors on unused parameters. \*/**

**// "noImplicitReturns": true, /\* Report error when not all code paths in function return a value. \*/**

**// "noFallthroughCasesInSwitch": true, /\* Report errors for fallthrough cases in switch statement. \*/**

**/\* Module Resolution Options \*/**

**// "moduleResolution": "node", /\* Specify module resolution strategy: 'node' (Node.js) or 'classic' (TypeScript pre-1.6). \*/**

**// "baseUrl": "./", /\* Base directory to resolve non-absolute module names. \*/**

**// "paths": {}, /\* A series of entries which re-map imports to lookup locations relative to the 'baseUrl'. \*/**

**// "rootDirs": [], /\* List of root folders whose combined content represents the structure of the project at runtime. \*/**

**// "typeRoots": [], /\* List of folders to include type definitions from. \*/**

**// "types": [], /\* Type declaration files to be included in compilation. \*/**

**// "allowSyntheticDefaultImports": true, /\* Allow default imports from modules with no default export. This does not affect code emit, just typechecking. \*/**

**"esModuleInterop": true /\* Enables emit interoperability between CommonJS and ES Modules via creation of namespace objects for all imports. Implies 'allowSyntheticDefaultImports'. \*/**

**// "preserveSymlinks": true, /\* Do not resolve the real path of symlinks. \*/**

**/\* Source Map Options \*/**

**// "sourceRoot": "./", /\* Specify the location where debugger should locate TypeScript files instead of source locations. \*/**

**// "mapRoot": "./", /\* Specify the location where debugger should locate map files instead of generated locations. \*/**

**// "inlineSourceMap": true, /\* Emit a single file with source maps instead of having a separate file. \*/**

**// "inlineSources": true, /\* Emit the source alongside the sourcemaps within a single file; requires '--inlineSourceMap' or '--sourceMap' to be set. \*/**

**/\* Experimental Options \*/**

**// "experimentalDecorators": true, /\* Enables experimental support for ES7 decorators. \*/**

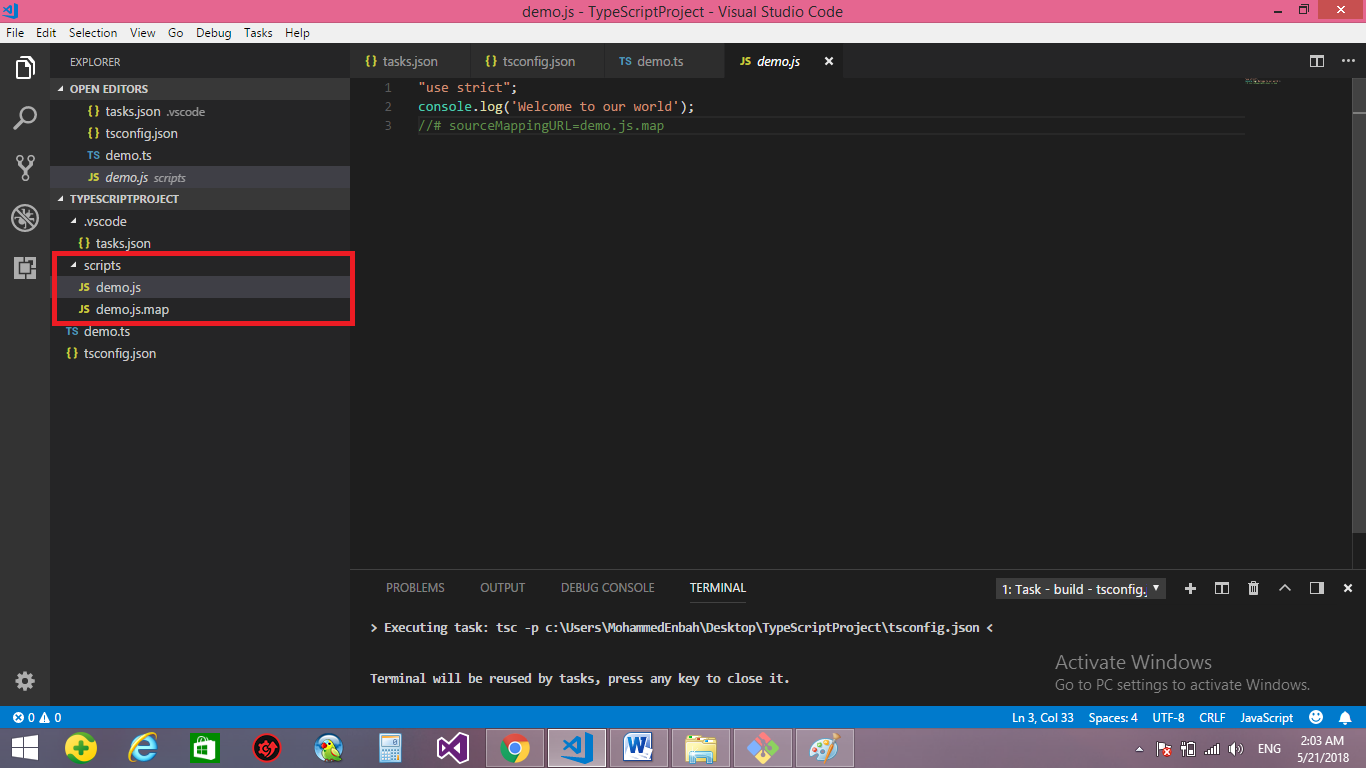
**// "emitDecoratorMetadata": true, /\* Enables experimental support for emitting type metadata for decorators. \*/**

**}**

**}**

**7-on the project 🡪 new file 🡪 call it as demo.ts 🡪 type the following command Console.log(‘hello world);Then save and press ctrl + shift +B**

**🡺 You will see that it create folder called script and inside it have the script file**

****

**8-on the folder 🡪 new file 🡪 type it as index.html and then type**

<html>

<head>

<title>Hello world</title>

</head>

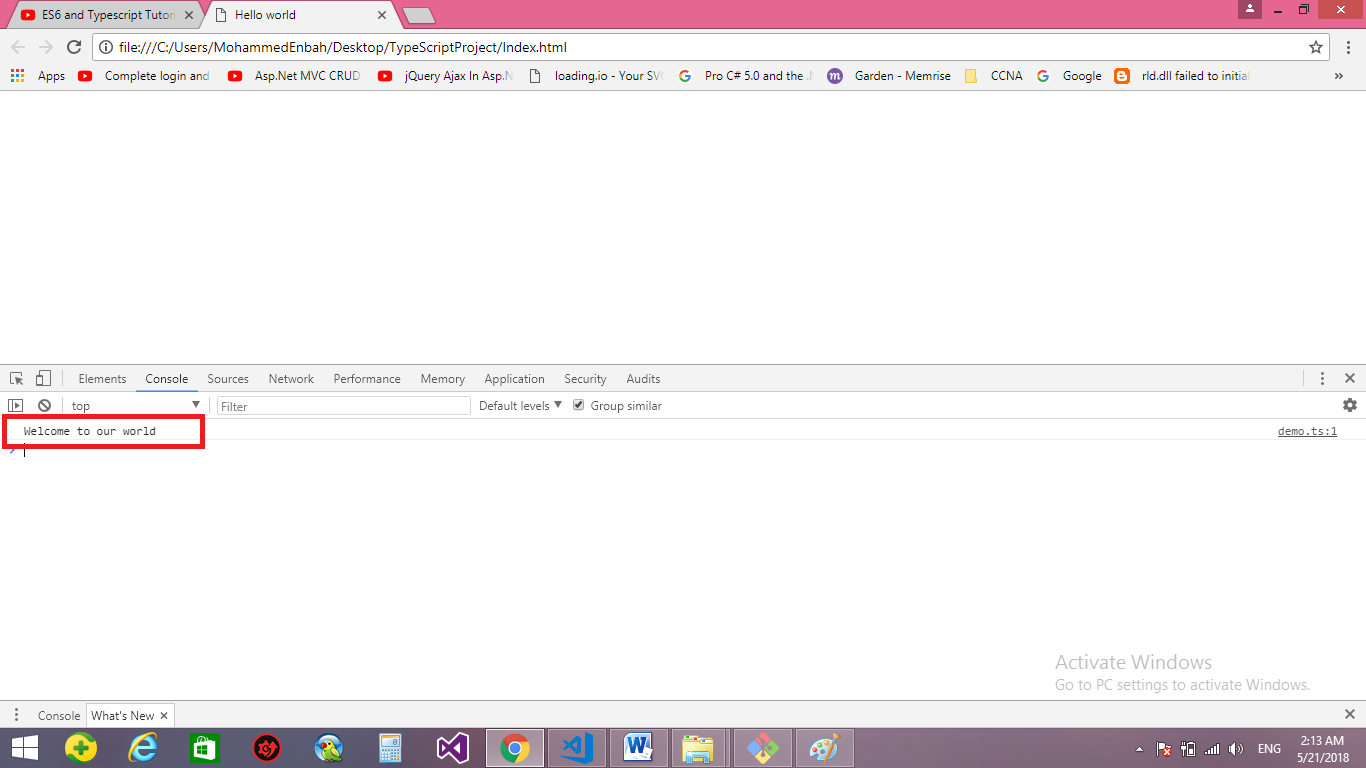
<body>

<script src='scripts/demo.js' type="text/javascript"></script>

</body>

</html>

**When you browse it will show the result in the console**

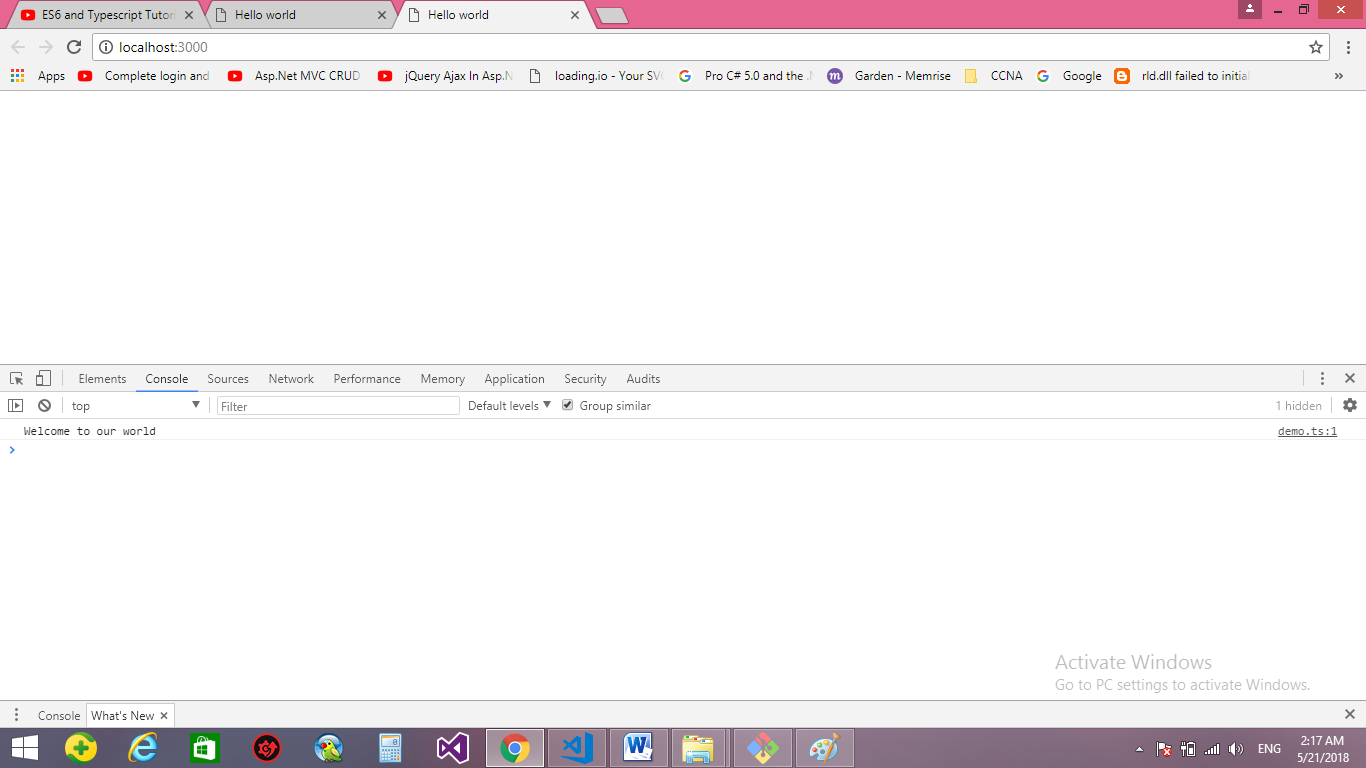
****

**file:///C:/Users/MohammedEnbah/Desktop/TypeScriptProject/Index.html**

**But we see that the url is shown the physical path , to resolve it we use the lite server and execute it on the cmd**

**Npm install lite-server –g //to install the lite server**

**Lite-server //to activate the lite server**

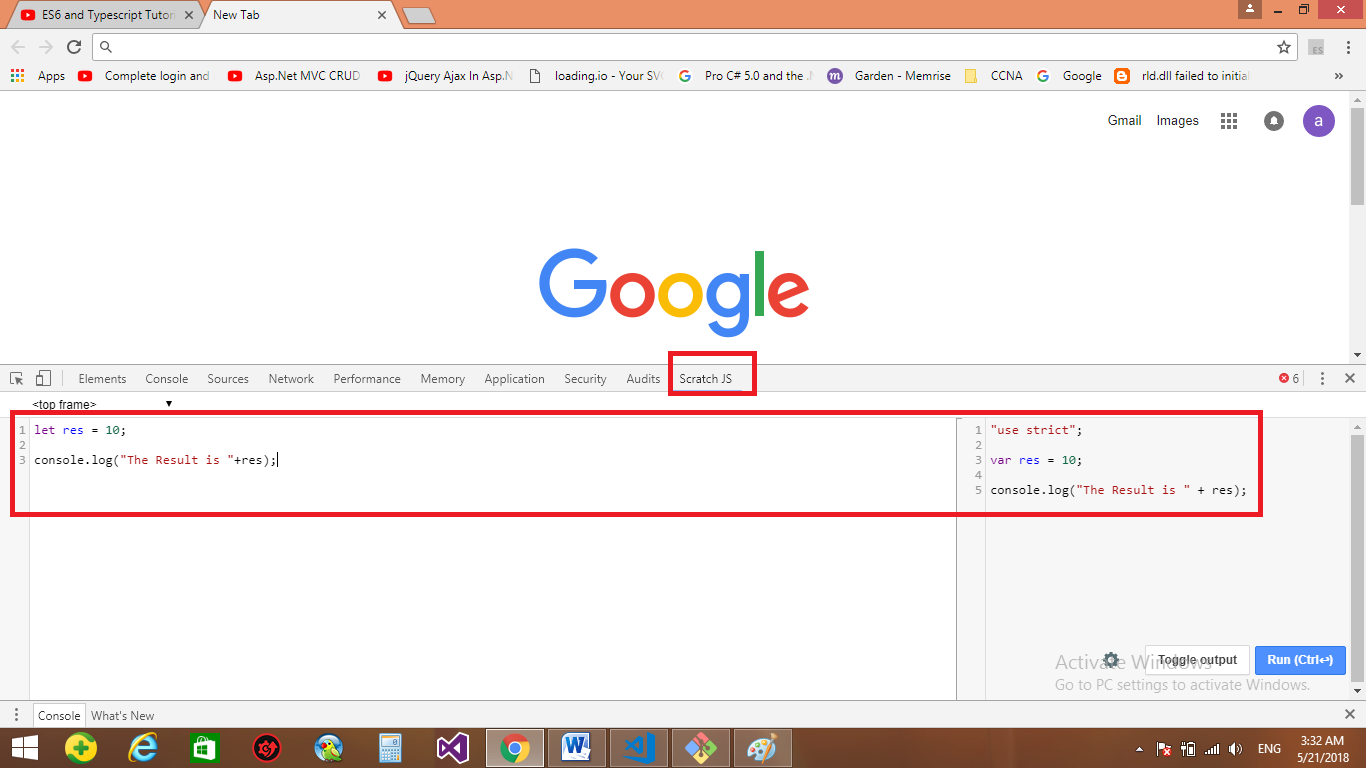


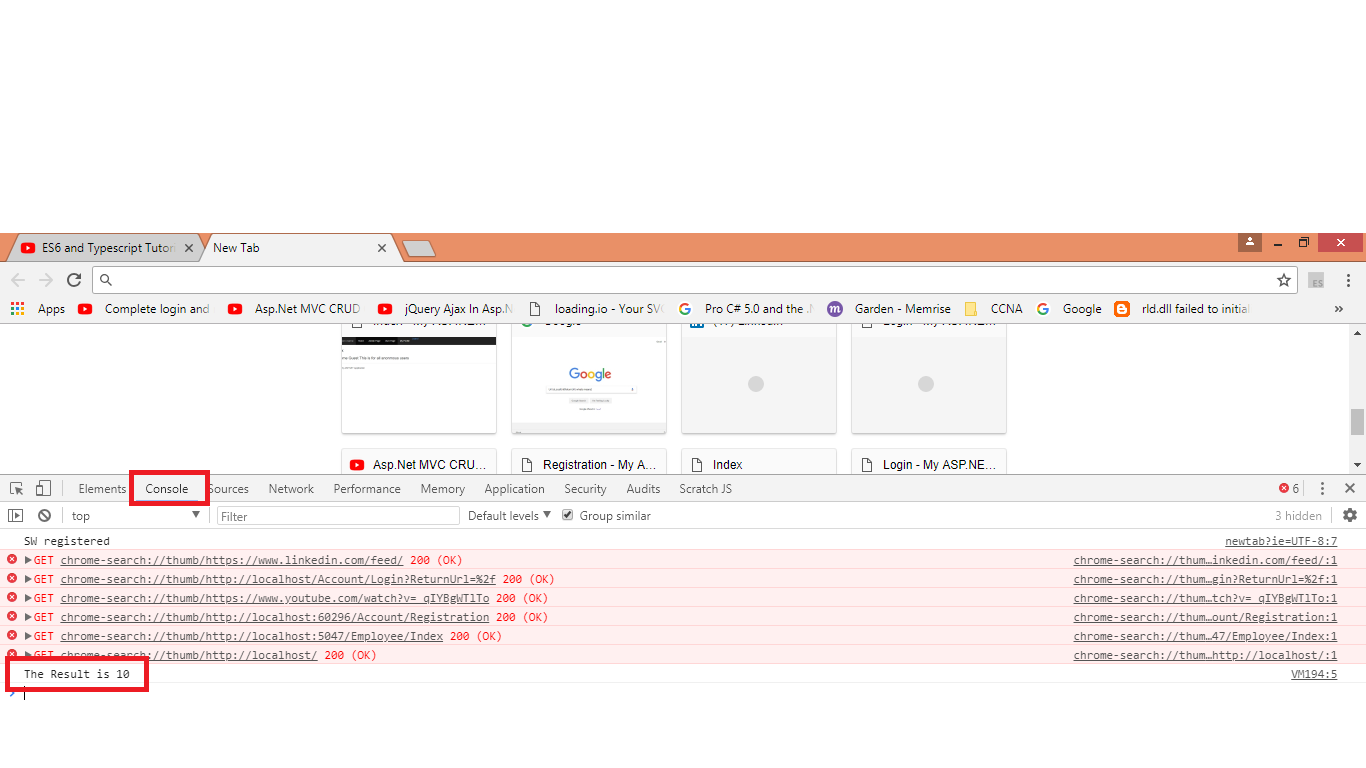
**Lesson04 Scratch JS**

**Scratch JS :- write and execute ES6/ES2015 within DevTools**

**This extension is an add-on for DevTools which integrates both the Traceur and Babel transpilers, allowing you to test out the new JS features coming with ECMAScript 6, also known as ES6 or ES2015. It also supports CoffeeScript, LiveScript and more compile-to-JS languages will be added soon.**

**🡺 You can with the Scratch JS to write Ecma 6 and to convert to Ecma 5**

****

****

**Lesson05 Hoisting & Functional Scope**

**Notes:-**

**When we using var keyword with variable it will happens Hositing**

**Hoisting: - means that any variable declared, the JavaScript put it in the top of the function**

**Functional Scope :- because the JavaScript Compiler put the variable definition on the top of the function , it will be seen across over all the function , so it does not matter that if you put the console.log inside or outside if stamen , because it will read the variable on the both 2 way**

**Example:-**

"use strict";

function CheckName(name) {

if (name == "Mohammed") {

var Msg = "Hello Mohammed";

}

else {

var Msg = "Hello Guest";

}

console.log(Msg);

}

CheckName("Mohammed1");

//# sourceMappingURL=demo.js.map

**The Compiler will put the variable Msg in the top of function to be like**

"use strict";

function CheckName(name) {

//it will make the variable in the top of the function

var Msg;

if (name == "Mohammed") { Msg = "Hello Mohammed";}

else { Msg = "Hello Guest"; }

console.log(Msg);

}

CheckName("Mohammed1");

//# sourceMappingURL=demo.js.map

**Even the local variable is put in the bottom of the function; the JavaScript internally put it in the top of the function**

**Lesson06 ES6 and Typescript, Let Keyword**

**Notes:-**

**1-resolve the hoisting, we can use let keyword that use block scope (not hoisted)**

function CheckName(name: string){

if(name == "Mohammed"){

let Msg = "Hello Mohammed";}

else{

let Msg = "Hello Guest";}

//we see that the variable is inaccessible because let keyword is block scope

console.log(Msg);}

CheckName("Mohammed");

**To resolve that we define the variable on the top of the function**

function CheckName(name: string){

let Msg:String;

if(name == "Mohammed"){

Msg = "Hello Mohammed";}

else{

Msg = "Hello Guest";}

console.log(Msg);}

CheckName("Mohammed");

**Example**

//we see that the var keyword is accessable inside and outside if statement

//we see that the let keyword is accessable only inside if or outside if

var a = 1;

var b = 2;

if(a == 1){

var a = 10;

let b = 20;

console.log(a); // 10

console.log(b); // 20

}

console.log(a); //10

console.log(b); //2

**The result will be = 10, 20, 10, 2**

**When we mean that the let not hoisted, we mean that we cannot declare the same variable name in the same block like this**

//This is not allowed

let x = 20;

let x = 30;

**//however if you made it as the below , it will make no problem because outside if is block and if is block and else is block**

function CheckName(name: string)

{

let Msg;

if(name == "Mohammed")

{

let Msg = "Hello Mohammed";

}

else

{

let Msg = "Hello Guest";

}

console.log(Msg);

}

**//and we cannot make declaration of variable on the end of function and use the variable before the declaration**

function CheckName(name: string)

{

if(name == "Mohammed")

{

Msg = "Hello Mohammed";

}

else

{

Msg = "Hello Guest";

}

console.log(Msg);

//it’s not allowed , unlike var keyword that is after debugging declare the variable on the top of the function internally

let Msg;

}

**Lesson07 Let keyword in for loop**

**The var keyword inside loop execute the commands and not wait the command to be executed then continue**

//in this example we see that the variable is increase and not wait the timeout to be finished

for(var i = 1 ; i<=5;i++)

{

setTimeout(() => {console.log(i)}, 1000);

}

So it will show only 6

//while if we use let keyword it will wait each sleep to execute so it will show all the numbers 1,2,3,4,5

for(let i = 1 ; i<=5;i++)

{

setTimeout(() => {console.log(i)}, 5000);

}

**Lesson08 const keyword**

**1-const is just like let keyword that is not hoisted and is for read only**

let x ;

//we have to initialize the value of the const variable

const no = 10;

//we cannot assign value to it

no = 20;

const emp = {name : 'Mohammed'};

console.log(emp.name);

//we cannot assign new object to it , however you can change the property value of it

emp = {}

emp.name = 'Isam'

console.log(emp.name);

**2-it’s prefer to use const when the reassignment not expected , while use let when the reassignment is expected**

const PI = 3.14;

const Max\_size = 100;

let x = 5;

let y = 10;

console.log('The Old Value Of X is ',x);

console.log('The Old Value Of Y is ',y);

x = x + y;

y = x - y;

x = x - y;

console.log('The New Value Of X is ',x);

console.log('The New Value Of Y is ',y);

**Lesson9 Arrow Function**

**Provide sharp syntax of function and simplify using of the function and assign to variable**

//The Standard way to assign function to variable

var Multi = function(fno:number,scno:number)

{

return fno \* scno;

}

console.log(Multi(10,5));

//if we make multiple lines , we have to use return keyword

var Mutli2 = (fno1:number,scno1:number) =>

{

return fno1 \* scno1;

}

console.log(Mutli2(2,3));

//if we make single line , we dont need to use return keywrod

var Multi3 = (fno:number,scno:number) => fno\*scno;

console.log(Multi3(4,5));

**Lesson10 this keyword in Typescript**

**Notes:-**

**1-The benefits of using Arrow Functions is that**

**A-Reduce the size of code**

**B-Resolve the use of using this inside function**

//The Standard Way to using this keyword inside function

var emp = {"id":3,

"Greet":function(){

console.log(this.id);

}}

emp.Greet();

//because each function has declare it’s own variable , we see that it’s make undefined , so the only solution for using the standard function is using variable and assign to the id

var emp = {"id":3,

"Greet":function(){

var x = this.id;

setTimeout(function(){console.log(x)}, 1000);

}};

emp.Greet();

//we see that with lamda expression it resolve the problem of using this keyword inside function if we make function inside function because lamda expression assign to the parent function not make it’s own function and it’s own variale

var emp = {"id":3,

"Greet":function(){

setTimeout(() => console.log(this.id), 1000);

}}

emp.Greet();

**Lesson11 Default Function Parameters**

**Notes:-**

**1-In Ecma 5 (JavaScript) , if you call function with parameter and don’t pass it parameter , it will tell you undefined**

let fun = function(v1:number)

{

console.log(v1);

}

//it will make undefined value

fun();

//we used default parameters

let fun = function(v1:number=10,v2:number=20)

{

console.log(v1+v2);

}

//the result is 10+20 = 30

fun();

//the result is 5+20 = 25

fun(5);

//the result is 30+2 = 32

fun(30,2);

//the result is 10+14 = 24

fun(undefined,14);

**//we can assign parameter with parameter operation**

let fun = function(v1:number=10,v2:number=v1\*0.1)

{

console.log(v1+v2);

}

//the result is 10+1 = 11

fun();

//the result is 5+0.5 = 5.5

fun(5);

//the result is 30+2 = 32

fun(30,2);

//the result is 10+14 = 24

fun(undefined,14);

**//we can assign external variable inside parameter**

let per = 0.1;

let fun = function(v1:number=10,v2:number=v1\*per)

{

console.log(v1+v2);

}

**//we can assign function inside parameter**

let per = () => 0.1;

let fun = function(v1:number=10,v2:number=v1\*per())

{

console.log(v1+v2);

}

**We can print the number of argument passed**

let per = () => 0.1;

let fun = function(v1:number=10,v2:number=v1\*per())

{

console.log(v1+v2);

//it will print the number of parameter passed

console.log(arguments.length);

}

//the result is 10+1 = 11

//0

fun();

//the result is 5+0.5 = 5.5

//1

fun(5);

//the result is 30+2 = 32

//2

fun(30,2);

//the result is 10+14 = 24

//2

fun(undefined,14);

**//we cannot assign second parameter as value to the first parameter**

let per = () => 0.1;

let fun = function(v1:number=v2,v2:number=v1\*per()){

console.log(v1+v2);

//it will print the number of parameter passed

console.log(arguments.length);}

**Lesson12 Rest Operators**

**Notes:-**

**1-if we pass to function number of parameter less or high than expected , it will agree on it**

let fun1 = function()

{

console.log(arguments.length);

for(let i in arguments)

{

console.log(arguments[i]);

}

}

//it will show 0

fun1();

//it will show 0

10

fun1(10);

//it will show 2

10

20

fun1(10,20);

**2-and if you want to separate parameters in group to make operate on them you cannot on the standard way**

let fun1 = function()

{

console.log(arguments.length);

for(let i in arguments)

{

console.log(arguments[i]);

}

}

//the function will operate the variable msg in the same argument array , so you cannot

//seperate on them

let msg:string = "Hello";

//the result will be 1

// hello

fun1(msg);

//the result will be 2

// hello

// 10

fun1(msg,10);

//the result will be 3

// hello

// 10

// 20

fun1(msg,10,20);

**3-the solution is using Rest that make separation into group of array and you can make operations as you want**

let fun1 = function(msg, ...colors){ //the yellow is called Rest it carry 3 item

console.log(msg);

console.log(colors.length);

for(let i in colors){

console.log(colors[i]);

}}

//With using …colors ,we make REST Operation , which means that separate the msg and the other parameter in named colors

//We won’t use argument , but argument is store all its variables on it , so we used the variable colors

let msg:string = "Hello";

//the result will be Hello

// 0

fun1(msg);

//the result will be hello

// 1

// 10

fun1(msg,10);

//the result will be hello

// 2

// 10

// 20

fun1(msg,10,20);

**Lesson13 Spread Operators**

**1-Rest is used to combine multiple parameters into single array**

**2-Spread is used to split the array into multiple parameters**

**3-Rest is called by expression … in the function only**

**4-Spread is called by expression … in the function and in call method too**

//we see that it call here

let fun1 = function(msg, ...colors)

{

console.log(msg);

for(let i in colors)

{

console.log(colors[i]);

}

}

//the function will operate the variable msg in the same argument array , so you cannot

//seperate on them

let msg:string = "List Of Colors";

let lst = ['yellow','red','green','black'];

//we see that it call here

fun1(msg,...lst);

**Lesson14 Object Literal 1**

**Notes:-**

**1-The Standard way of define object in JavaScript**

let Fname = "Mohammed";

let Lname = "Ali";

//This is the standard way to define object in javascript

var emp = {

Fname : Fname,

Lname : Lname

};

console.log(emp.Fname);

console.log(emp.Lname);

**2-In Typescript is the way to define the object**

let Fname = "Mohammed";

let Lname = "Ali";

//In TypeScript we can now define the object like this

var emp = {

Fname,

Lname

};

console.log(emp.Fname);

console.log(emp.Lname);

**3-we can make function inside the object decleration**

function CreateObject(fname:string,lname:string,age:number){

let fullname = fname + " "+lname;

return{fullname,

lname,

fname,

//We can define function inside property in the object declaration

isSenior:function()

{return age > 50;}

}}

var emp = CreateObject("ALi","Osama",65);

console.log(emp.fname);

console.log(emp.lname);

console.log(emp.fullname);

console.log(emp.isSenior());

**Lesson 15 Object Literal Part 2**

let ln = "Last Name";

let emp =

{

//we can make property with space with one condition with using ""

"First Name":"Mohammed",

//we can use the varaible outside the object with [] syntax

[ln] : "Enbah"

}

//the call in the 2 way like this

console.log(emp["First Name"]);

console.log(emp["Last Name"]);

**Lesson16 Destructing Array in Typescript**

**Notes:-**

**1-Destructing Array: - is the process that split the array of item and assign it’s values to the variables**

let lst = ["Mohammed","Enbah","Male"];

//The Destructing operation happens when destruct the array into multiple objects

let [Fname,Lname,Gender] = lst;

console.log(Fname);

console.log(Lname);

console.log(Gender);

//The Result will be

//Mohammed

//Enbah

//Male

//you can implement Rest Operatior in destrucing array

let [Name,...l1] = lst;

console.log(Name);

console.log(l1);

//The Result will be

//Mohammed

//{"Enbah","Male"}

let lst1 = ["Mohammed","Enbah"];

//you can implement default value in the destructing array

let [fname,lname,gender="Female"] = lst;

console.log(fname);

console.log(lname);

console.log(gender);

//The Result will be

//Mohammed

//Enbah

//Female

//Note if you pass number of variable more than number of items on the

//array the value of the variables is undefined

**Lesson17 Destructing Objects**

let employee =

{

fname : "Mohammed",

Lname : "Enbah",

Gender : "Male"

}

//when we make destructing make sure that the variable name match the property name

let {fname,Lname,Gender} = employee;

console.log(fname);

console.log(Lname);

console.log(Gender);

//if you want to use differnt name , you can use alias name

let{fname:f,Lname:l,Gender:g} = employee;

console.log(f);

console.log(l);

console.log(g);

**Lesson 18 String Template**

//This is the standard way to used contactication

var oldname = "Ali Ahmed";

var oldfullname = "Your Full Name is "+oldname;

console.log(oldfullname);

//There is new technique called string template with

//using the character ` not ' , the ` exist above tab button

var name1 = 'Mohammed Enbah';

var Fullname = `Hi ${name1} , I hope you fine`;

console.log(Fullname);

//you can use the string template over multilines , and the spaces are taken as inserted

var text = `hello ${name1} , this is wonderfull world

,Right where is your brother pro

, ha ha ha , i will kill you ,

, just jokking`;

console.log(text);

**Lesson19 for loop**

let colors = ["Red","Orange","Blue","Green","Yellow"];

//we have two type of for loop

//1-for in ==> similar to for(int i = 0 ; i< = lst.legnth;i++){console.log(lst[i])}

//2-for out ==> similar to foreach(var color in colors){console.log(color)}

//for each looks to variable as list of string , list of characters = string

for(let i in colors){

console.log(colors[i]);}

//it's look it as array of string

for(let color of colors){

console.log(color);}

let name\_first = "Mohammed";

//it's look it as array of characters

for(let chr of name\_first)

{

console.log(chr);

}

**Lesson20 Class**

**Notes:-**

**1-Class is special function but it’s not hoisted , this is means that if you declare variable before the class declaration this is make error unlike any normal function**

//if we declare the method before or after method declaration, it’s not make any error for that

operate();

function operate()

{

console.log("Code Activate")

}

operate();

//if we declare object before class declaration it will generate error message

let emp = new employee();

class employee

{

public Show()

{

console.log("Hi There");

}

}

let emp1 = new employee();

emp1.Show();

**Lesson21 Class Component**

**1-there are 3 types of methods on the class**

**A-Constructor: - is activate when the object is initialized**

**B-Static Method: - is called without the object initialize**

**C-Normal Method: - is called with the object initialize**

class employee

{

fname:string;

//we have to know that the field must be assigned value or called on the constructor

// lname:string;

constructor(fname:string)

{

this.fname = fname;

console.log("Constructor Is Called");

}

static StaticMethod()

{

console.log("Static Mehtod Is Called");

}

NormalMethod():void

{

console.log("Normal Method Is Called");

}

}

let emp = new employee("Mohammed");

emp.NormalMethod();

employee.StaticMethod();

**Lesson22 Class Inheritance**

class Person{

constructor(protected fname:string,protected lname:string){

console.log("The Person is created ");}

public ShowPersonFullName(){

console.log("The Person Full Name is "+this.fname+" "+this.lname);}}

class employee extends Person{

constructor(protected fname:string,protected lname:string){

super(fname,lname);

console.log("The Employee is created");}

public ShowEmployeeFullName(){

super.ShowPersonFullName();

console.log("The Person Full Name is "+this.fname+" "+this.lname);}}

let emp = new employee("Mohammed","Enbah");

emp.ShowEmployeeFullName();

**Lesson 23 Modules**

**1-Modules is Separation of concern**

Module

C

Module

B

Module

A

**2-we can use modules with import and export operation to take what you want to the Module**

Module B

Function

export

Module A

Import

**Lesson 24 Module Loader Setup**

**\*To install module loader , follow these 3 steps**

**1-on your project create new folder called npm\_module**

**\*make on the cmd the following command**

**Cd ~/Desktop/TypeScriptPro/npm\_module**

**//to installs the Module Loader**

**Npm install es6-module-loader traceur systemjs**

**\*note:we see that it create folders required to loading your modules**

**2-create folder on your project called src and create 2 JavaScript folders ModuleA.js and ModuleB.js**

**3-on your html page write the following code to reference the 3 folder required to loading your modules and import the JavaScript file**

<html>

<head>

<title>Hello world</title>

<script src="module\_node/node\_modules/traceur/bin/traceur.js"></script>

<script src="module\_node/node\_modules/systemjs/dist/system.src.js"></script>

<script src="module\_node/node\_modules/es6-module-loader/dist/es6-module-loader-dev.js"></script>

</head>

<body>

<script>

System.import('src/ModuleA.js');

</script>

</body>

</html>

**Module**

**1-Is group of classes or interfaces that the target is the grouping and there is no anonymous code, everything is sorting.**

**2-Moduels executed on own scope , not the global scope , this is means that the variables defined is not visible outside the module unless we use the import command**

**Types:**

**1-Internal Module (Namespace):- inside the typescript folder**

**2-external Module: - Outside the typescript folder**

Simple example of create anonymous object and then call the method inside it

var person ={

Fname: "Mohammed",

Lname: "Enbah",

GetFullName: function (){

console.log(person.Fname + " " + person.Lname);}}

person.GetFullName();

//create EmployeeContract.ts and then make the following code

export {IHomeContract,ICarContract}

interface IHomeContract{

FirstHomePart: string;

SecondHomePart: string;

ShowHomeContract():void;}

interface ICarContract{

FirstCarContract: string;

SecondCarContract: string;

ShowCarContract():void;}

//create EmployeeClass.ts and then execute the following code

//This is the way to import module

import c1 = require("./EmployeesContract")

class employee implements c1.IHomeContract,c1.ICarContract{

FirstCarContract: string;

SecondCarContract: string;

ShowCarContract(): void {

console.log("The First Car Contract is "+this.FirstCarContract+"\n and the second Car Contract is "+this.SecondCarContract)}

FirstHomePart: string;

SecondHomePart: string;

ShowHomeContract(): void {

console.log("The First Home Contract is "+this.FirstHomePart+"\n and the second Home Contract is "+this.SecondHomePart)}

constructor(firstcarpart: string = "", secondcarpart: string ="",firstHomeContract: string = "",secondHomeContract:string = ""){

this.FirstCarContract = firstcarpart;

this.SecondCarContract = secondcarpart;

this.FirstHomePart = firstHomeContract;

this.SecondHomePart = secondHomeContract;}}

let emp = new employee("Mohammmed","Enbah");

emp.ShowHomeContract();

**Module Loader**

**There are different types of module loader available like**

**CommonJs**

**SystemJs**

**AMD**

**UMD**

**ES2015 // the most popular Module Loader to load the module and execute the modules on the browsers**

**Why Module Loader?**

**1-The Module Loader are used to resolve the problem of loading multiple modules from different locations before the methods called**

**2-Module Loader are fast / asynchronous and they don’t bog down you application loading time**

**Some of Common web applications module loader are**

**1-Require Js**

**2-Browserify**

**3-Webpack**

**You see the module loader type on the tsconfig.json**

**1-In Task1 write the following code**

import {fname,lname} from './Task2' // you can import more than 1 type in scope

console.log(` ${fname} ${lname}`);

**2-In Task2.ts write the following code**

export {fname,lname} // you can put more than type into export

let fname:string = "Mohammed";

let lname:string = "Ali";

**3-You can make alias name for the variables**

import {fname as l1,lname as l2} from './Task2'

console.log(` ${l1} ${l2}`);

**4-Moduels are hoisted**

//In Task1.js

console.log("Task 1 is loaded");

import {fname as l1,lname as l2} from './Task2'

console.log(` ${l1} ${l2}`);

//In Task2.ts

let fname:string = "Mohammed";

let lname:string = "Ali";

console.log("Task 2 is loaded");

export {fname,lname}

//we see that when execute import it’s execute everything inside Task2 even the conosole.log

Task 1 is loaded

Task 2 is loaded

Mohammed Ali

**5-Import are read only**

//In Task1.js

console.log("Module 1 is loaded");

import {fname as l1,lname as l2} from './Task2'

//you see that the variable cannot change it's value

l1 = "Isam";

console.log(` ${l1} ${l2}`);

**//so the solution is make object and put it property**

//In Task2.ts

//we put the variables inside object and make it export

let obj = {

fname : "Mohammed",

lname : "Ali" }

console.log("Task 2 is loaded");

export {obj}

//In Task1.js

console.log("Module 1 is loaded");

import {obj as emp} from './Task2'

//you see that the property fname , lname can change it’s value

emp.fname = "Isam";

emp.lname = "Mosa";

console.log(emp.fname + " "+emp.lname);

**Lesson27 Default Export**

**1-If you want to pass single variable or function, you can use Default Export**

//In Task2.js

let fname = "Mohammed";

//if you want to export single variable or function

export default fname;

//In Task1.js we see that the import vairable can be differnt from the export

import firstname from './Task2'

console.log(firstname);

**//you can use alias name for the default export**

//In Task1.js we see that the import vairable can be differnt from the export

import firstname from './Task2'

console.log(firstname);

**Lesson28 Export Function and classes**

//In Task2.ts we do export operation

export{DoOperate,Employee}

function DoOperate(msg:string)

{

console.log(msg+" is called ");

}

class Employee

{

constructor(public fname:string,public Lname:string)

{

console.log("The Employee Is Created");

}

ShowEmployeeInfo():void

{

console.log("The Employee Full Name is "+this.fname+" "+this.Lname);

}

}

//we make import operation

import {DoOperate,Employee} from './Task2'

DoOperate("Hello World");

let emp = new Employee("Mohammed","Ali");

emp.ShowEmployeeInfo();

**Lesson29 Set and Map**

let emp = Object.create(null);

emp.id = true;

//to check that the value is exist or not we use this expression

if(emp.id != null){

console.log('Employee ID exist');}

else{console.log('Employee ID Not exist');}

let myMap = Object.create(null);

myMap.name = "Moahmmed";

let val = myMap.name;

console.log(val);

//we see that the mapping theory that if the 2 variable of the same type and assign value to one of them

//and we want to assign value to another variable , the mapping will put the value also on the another vairable

let l1 = {};

let l2 = {};

myMap[l1] = "Ali";

console.log(myMap[l2]);

console.log(myMap[l1]);

**Lesson29 Set keyword**

**Set is used to store unique values only inside it and use only size and has to check if the value exists or not**

**(If you want to make trace if the values exist, you can use Set keyword)**

**(If you want to make trace if the objects exist, you can use WeakSet keyword)**

let lst = new Set();

//we can store any data type becouse it's object data type

let l1 = {};

let l2 = {};

lst.add(1);

lst.add("Mohammed");

lst.add(true);

lst.add(l1);

lst.add(l2);

console.log(lst.size);

//The Set Keyword is store only primitive values , this is means that it does not accept duplicate values

let lst1 = new Set([1,2,3,4,5,5,5]);

//it will print 5

console.log(lst1.size);

lst.add("Hello1").add("Hello2");

console.log(lst.size);

//to check that the item exist in the list

console.log(lst.has(l1));

//to delete specific item in the list

lst.delete(l1);

console.log(lst.has(l1));

**Lesson30 Weak Set**

**1-Strong Set is accessible for the variables and not accessible to the object assignment it’s accessible to the instance that assign to the object**

**In the following example ,we see that the instance is the only care for the Set() , if we set the object that assigned by instance , the result not effect**

let lst = new Set();

let emp = {};

lst.add(emp);

//we see that the size is 1

console.log(lst.size);

//we see that even the instance is cut connection with the object , we see that the size is 1

emp = null;

console.log(lst.size);

//emp = [...lst][0];

//To Resolve the Object Reference issue ,we use Weak Set

//The Only B1enefit of the WeakSet is to trace the object in memory to know if the object exist or not

let weaklst = new WeakSet();

let weakemp = {};

weaklst.add(weakemp);

console.log(weaklst.has(weakemp));

//we see that the size is undefined

console.log(weakemp.size);

weakemp = null;

console.log(weaklst.has(weakemp));

//console.log(weakemp.has(weakemp));

**Lesson31 Maps**

**Store as the principle in key ,value pairs**

let lst = new Map();

lst.set("Fname","Mohammed");

lst.set("Age",21);

console.log(lst.get("Fname"));

console.log(lst.get("Age"));

//you can assign variable in the item on the list

let l1 = {};

let l2 = {};

lst.set(l1,"List 1");

lst.set(l2,"List 2");

console.log(lst.get(l1));

console.log("Before Delete Items is "+lst.size);

lst.delete("Age");

console.log("After Delete Item is "+lst.size);

lst.clear();

console.log("After Clearing List is "+lst.size);

**Lesson32 Iterating with Map**

**You can make iteration of the keys, values, or both**

//This is the way to store array of keys and values inside the Map Variable

let lst = new Map([

["Fname","Mohammed"],

["Lname","Ali"]

]);

console.log(lst.size);

//you can get the keys

for(let key of lst.keys())

{

console.log(key);

}

//you can get the values

for(let value of lst.values())

{

console.log(value);

}

//you can get the both value and key

for(let entry of lst.entries())

{

console.log(`${entry[0]} -> ${entry[1]}`);

}

//if you want to destructring the Map , you can do this by the below code

for(let [key,value] of lst.entries())

{

console.log(`${key} --> ${value}`);

}

**Lesson33 Make Iteration with Set, Array, Map**

let lst = [1,2,3,4,5];

//we can make iteration of the Array like the below code

lst.forEach(MakeIteration);

function MakeIteration(element,index,array){

console.log(`arr[${index}] => ${element}`);}

let MapLst = new Map([

["Fname","Mohammed"],

["Lname","Enbah"]]);

//we can make iteration of the Map like the below code

MapLst.forEach(MakeMapIteration)

//The CallingMap is mean the set variable inserted

function MakeMapIteration(value,key,CallingMap){

console.log(`${key} => ${value}`);

console.log(`${CallingMap == MapLst}`);}

//we can make iteration with set like the below code

let SetLst = new Set([1,2,3,4,5]);

SetLst.forEach(MakeSetIteration);

//The CallingSet is mean the set variable inserted

function MakeSetIteration(value,key,CallingSet){

console.log(`${key} => ${value}`);

console.log(`${CallingSet == MapLst}`);}

**Lesson35 Weak Map**

**Like the Weak Set, it can use for tracing objects**

let lst2 = new Map();

let l2 = {};

lst2.set(l2,"Ali");

console.log(lst2.get(l2));

l2 = null;

console.log(lst2.get(l2));

//for object References We Using the Weak Map

let lst = new WeakMap();

let l1 = {};

lst.set(l1,"Mohammed");

console.log(lst.get(l1));

l1 = null;

console.log(lst.get(l1));

**Lesson 36 Symbol**

**To make Unique Id that can used inside object and functions**

//we see that even the Description is the same , the ID is not equal

let s1 = Symbol('Test');

let s2 = Symbol('Test');

console.log(s1 === s2);

//The Only way to match the ID is by using Symbol.for

let s3 = Symbol.for('Test');

console.log(s1 === s3);

console.log(Symbol.keyfor(s1));

console.log(Symbol.keyfor(s2));

console.log(Symbol.keyfor(s3));

**Lesson37 Symbol.iterator**

**1-To Know that the data type is valid to use in the for of loop , we use the Symbol.iterator**

let lst = [1,"w",54,true]

let str = "Mohammed";

let no = 100;

let obj = {fname :"Ali Ahmed"};

//we see that the string (Array of character) and Array return function

console.log("Array work on for loop to get values "+typeof lst[Symbol.iterator]);

console.log("String work on for loop to get values "+typeof str[Symbol.iterator]);

//we see that the object and number return null

console.log("Number work on for loop to get values "+typeof no[Symbol.iterator]);

console.log("Object work on for loop to get values "+typeof obj[Symbol.iterator]);

# Lesson38 Iterables and Iterators

# 1-Iterables :- is any object that implement method who keys is symbol.iterator

# 2-Iterator :- is an object who implement the next method

# Next method : how to access element in array , Map , Set , ….

//we see that the Iterable contain the method of type Symbole.Iterator that return Iterator

Iterable

{

[Symbol.Iterator]() : Iterator

}

//we see that the Iterator have method next()

Iterator

{

next() : IResultObj

}

//we see that the IResultObj has 2 component :

//value : any ===> carry any data type , done : bool ==> means that the Iteration is complete

IResultObj{

value : any

done : bool}

# We can make do our Iterator on this code

let lst = [1,2,3];

function CreateIterator(array){

let count = 0;

return{

//the next return object with 2 property value , done

next:function(){

return count < array.length?

{value:array[count++],done:false}:

{value:undefined,done:true}

}}}

//it will fill the lst inside the MyIterator

let MyIterator = CreateIterator(lst);

//we can access the Iterator items by using next()

console.log(MyIterator.next());

console.log(MyIterator.next());

console.log(MyIterator.next());

console.log(MyIterator.next());

# Lesson 38 Object Iterator

# 1-By Default the Array and string are valid for Iterator , but object and number are not valid to Iterator , if you want to make Custom Iterator to the object , you have to make custom Symbol.iterator

let person =

{

Fname : "Mohammed",

Lname : "Enbah"

}

person[Symbol.Iterator] = function(){

let properties = Object.keys(person);

let count = 0;

let isDone = false;

let next = () =>

{

if (count >= properties.length)

{

isDone = true;

}

return{done : isDone,value :this[properties[count++]]};

}

return {next};

}

for(let item of person)

{

console.log(item);

}

# Lesson 40 Generators

# Generators: - is special function capable to pause function and then resume it using the yield keyword

//we see that the yield the basics is the same of the Symbol.Iterator

//this is means that when we call it with using the next() it will search for

//the next yeild and return object {value : 1 , done : false}

//and if not exist any more yeild it return {value : undefined , done : true}

function \*CreateGenerator(){

//after executing yield it will pause executing, and to return execute , we use //next()

yield 1;

console.log('After 1st yield');

yield 2;

console.log('After 2nd yield');

}

let myGene = CreateGenerator();

console.log(myGene.next());

console.log(myGene.next());

console.log(myGene.next());

//we see the result is :-

{value :1 , done : false}

After 1st yield

{value :2 , done : false}

After 2nd yield

{value :3 , done : true}

//we are using the yield to make pause the collecting more than one porperty and then call the second in the next loop

let person = {

Fname : "Mohammed",

Lname : "Enbah"}

person[Symbol.Iterator] = function\*(){

let properties = Object.keys(person);

for(let t in properties){

yield this[t];}}

for(let item of person){

console.log(item);}

# Lesson 40 Typescript

# -Typescript is super set of JavaScript

# -Typescript are optional

# Why Using Typescript?

# 1-Identify mistakes during development

# 2-Tool Support

# 3-Ts = ES5 + ES6 + ES7 + Features

# Lesson41 Declaration and Annotation

//By Default the var keyword will define the data type automatically based on it's value inserted

var x ="Mohammed";

var y = 20;

var z = true;

//we see that when assign value of different data type , it will make error message (This is called Type Inference)

x = 20;

//we will make explicit declration of the variable data type

var w:string = "Isam";

console.log(x);

# Lesson42 Type Inference

//By Default the var keyword will define the data type automatically based on it's value inserted

var x ="Mohammed";

//we see that when assign value of different data type , it will make error message

//(And This is Called Type Inference)

x = 20;

//we see that the data type is string , becouse the + operator is contatication

//so it's better to use explicit annotation rather the Type Inference

var w = x+"Isam";

console.log(x);

# Lesson43 Any Type

# 1-Any Type is the based type for the all type

# 2-By using Any Type , There is no compile time checking

var x1 : any;

x1 = 10;

x1 = "Ali";

x1 = true;

# Lesson44 Enumeration in Typescript

enum EyeColor {brown = 4,blue = 100,Green = 200};

var MyEyeColor = EyeColor.blue;

//This will store the key of the ENUM

console.log(MyEyeColor);

//This will store the value of the item in enum

console.log(EyeColor[MyEyeColor]);

# Lesson45 Array in Typescript

//There is 2 way to define the Array's in typeScript

let strlst : string[] = ["Ahmed","Mohammed"];

let intlst : Array<string> = ["Osama","Ahmed"];

for(let item of strlst)

{

console.log(item);

}

for(let item of intlst)

{

console.log(item);

}

//if you want to make array of specified 2 data type only , you can use the tupple array

let tupplelst : [string,number] = ["Isam",20]

tupplelst[2] = 500;

for(let item of tupplelst)

{

console.log(item);

}

# Lesson46 Constructors in Class

class Person

{

public Fname:string;

public Lname:string;

constructor(fname:string,lname:string)

{

this.Fname = fname;

this.Lname = lname;

}

}

//This is the shortest way to define constructors with fields

class Person1

{

constructor(public Fname:string,public Lname:string){}

}

# Lesson47 Interface

interface employee

{

fname:string;

lname:string;

//we can make optional property , this is means you can discard on the object decleration

Age?:number;

}

let emp1:employee =

{

fname:"Mohammed",

lname:"Enbah"

}

# Lesson48 TextDecerator

{

"compilerOptions": {

/\* Basic Options \*/

"target": "es5", /\* Specify ECMAScript target

"module": "commonjs", /\* Specify module code generation:

"sourceMap": true, /\* Generates corresponding '.map'

// "outFile": "./", /\* Concatenate and emit output to

"strict": true, /\* Enable all strict type-checking

"esModuleInterop": true , /\* Enables emit interoperability

"experimentalDecorators": true, /\* Enables experimental support for

}

}

# Text Decorators :- is special decleration that attached to classes , interfaces , and can evaluated by function at runtime

@Component({

selector : 'my-app',

template : '<h1>Hello World</h1>'

})