ANGULAR-15

Q. What is Angular?

Angular is a developers platform. Developers platform means Angular provides support from end to end i.e provides support from building to deployment. Angular provides all the languages, library & tools required for developer.

Q. What are the challenges in modern web development?

Generally smart devices have lower bandwidth.

Unified UX(User Experience): To build a website that is compatible/run in every devices. It is not as much easy task.

Fluid UX: It means the user don't need to go every pages. The user will stay in one page and access all the things in the same page.(SPA)

Loosely coupled and Extensible: Loosely couples means we can add new things without affecting the existing things.

The only solutions for this is Building SPA(Single Page Application).SPA has all the features like Unified UX, Fluid UX.

E-bay & Twitter are the first single page applications.

Q. Problems with JS & jQuery?

Lot of DOM Manipulations, Heavy on application, Lot of AJAX explicitly & Lots of coding.

The Solution for this building SPA using Angular, React, Vue, Backbone, knockout etc.

Q. Why we need Angular?

To build single page applications only. It is a Java script framework which reduces the efforts of developers. It provides language, libraries & tools required to develop applications.

Q. What is difference between Angular and Angular JS?

ANGULAR JS:

- Google introduced Angular JS in 2010 for building SPA.
- Angular JS is open source and cross platform framework developed by google and maintained by a large community of developers and organizations.
Angular JS is not designed for what you are using. Hence lot of GAPS(Gaps means the technology cannot fulfil all the needs).
- Angular JS uses JavaScript as Language.
- JavaScript is not strongly typed language.
- JavaScript is not an OOP language.
- Extensibility issues.
- Not Loosely Coupled
- Google Developed Angular JS versions Up to "1.8"
- Google Introduced "Angular" in 2014 as Version 2

ANGULAR-2:

- Google re-built complete technology as "Angular 2"
- Angular 2 is not a replacement for Angular JS, It is just as alternative.
- Angular 2 Features
  a) It uses "TypeScript" as a language.
  b) TypeScript is an OOP language.
  c) Modular Library(Lightweight Library)
  d) Light weight
  e) Faster
  f) Reduced GAPS
- Angular Latest is "15"

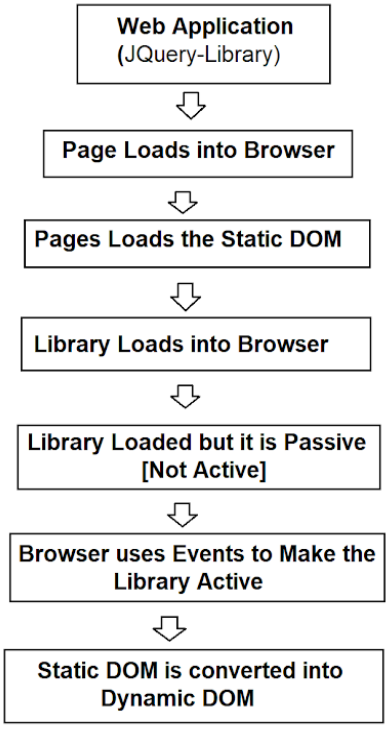
Q. What is difference between React and Angular?

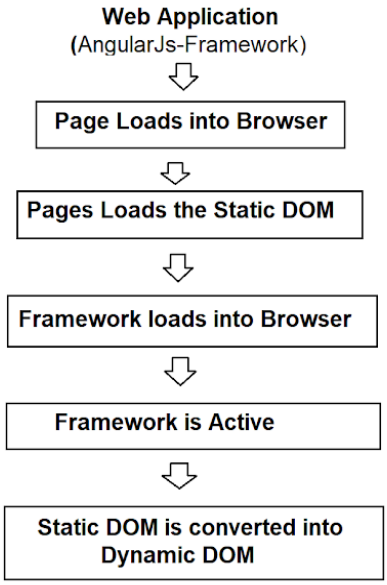
@ React is a Frontend library is used to building UI whereas Angular is a frontend framework is used for building Ui and controlling the Application flow.

@ React uses one way databinding and supports virtual DOM whereas Angular uses two-way data biding and supports real DOM.

@ React is faster than Angular because of smaller bundle size.

@ React provides support for adding Java script libraries to source code where as Angular JS does not provides.





Library and Language can't control application flow

Q. What are the Issues with Object Oriented Programming.

OOP does not support low level features. It cannot directly interact with hardware. Need more memory, Complex, Heavy, Slow.

Q. What are the issues with JavaScript?

- Not Strongly Typed Language.

```
var x = 10;
x = "John";
```

- Not Strictly Typed, It is explicitly strict.

```
x = 10;      // valid
"use strict";
```

- It is not an OOP language
- It supports only few features of OOP.
- It is hard to extend.
- It will not support code level security.
- Google started a language called "AtScript" for Angular
- Microsoft have TypeScript
- Microsoft .NET language "C#" Anders Hejlberg
- TypeScript Anders Hejlberg

** TYPESCRIPT **

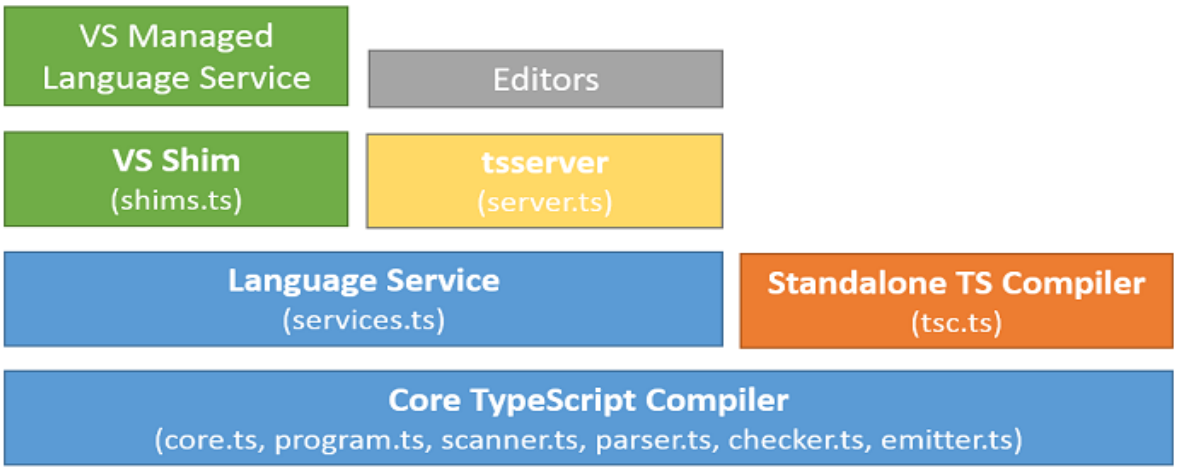
SUMMARY

Introduction to TYPESCRIPT

- It is a super set of Java script. It is strongly typed.
- It is strictly typed.
- It is an OOP language.
- It is built in TypeScript. TypeScript uses compile time checking.
- It can directly interact with hardware.
- Less memory, Faster, Light Weight
- It is used in building large scale applications.

Note: Browsers can't understand TypeScript So, TypeScript is translated into JavaScript.

TYPESCRIPT ARCHITECTURE



1. Core Compiler

- Core compiler translates the typescript code into machine code.(Verifies The Error in the program)
  - core. ts : It verifies the keywords
  - program. ts : It verifies the program structure. Check the syntax whether semicolon is misses or not & check other syntax also.
  - parser. ts : It is responsible for converting one type to another
  - checker. ts : It is responsible for data type checking.
  - scanner. ts : It is responsible for handling input.
  - emitter. ts : It is responsible for handling output.

2. Standalone Compiler [tsc.ts]

- It trans compiles Typescript program into JavaScript.

3. Language Service [services.ts]

- It provides pre-defined functions and values required for typescript.
- Service is a pre-defined business logic.

4. TsServer [server.ts]

- Hosting(Putting our developed application on Internet)
- Request and Response
- TypeScript programs are hosted, compile and managed on server.

5. VS Shim [shims.ts]

- It makes the typescript program cross platform. (cross platform means the software /program can able to run in any OS like Windows, MacOS, Linux)
- TypeScript Program => Compiler => IL Code => Shim => Managed Code
- Managed code is suitable for all OS services.

6. Manage Language Service

- It is the service suitable for all OS.
- It is cross platform.
- Platform neutral.

Note: CAPTCHA is developed through AI. A Java developer cannot develop a CAPTCHA. CAPTCHA is used to verify whether you are a human or Robot.

Editor is a platform where we write our program, compile and run our program.

SETUP ENVIRONMENT FOR TYPESCRIPT

1. Download and Install Node JS on your PC

- We are installing NodeJS on PC for a "Package Manager" called NPM.
- Package Manager is a tool used by developers to download and install various libraries required for project.
- Various package manager tools are
  NPM, Yarn, NuGet, Bower, Ruby-Gems etc

https://nodejs.org/en/

- Download 18.12 [LTS] Version

2. Check the version of NodeJS and NPM on your PC

```
C:\>node -v
C:\>npm -v
```

Note: Make sure that your Node version is > 14 and NPM version > 6.

3. Download and Install "Visual Studio Code" Editor

- Editor provides an IDE [Integrated Development Environment]
- You can build, debug, test, and deploy applications.
  - VS Code
  - Sublime
  - WebStrom
  - Notepad++ etc..

https://code.visualstudio.com/
http://editorconfig.org/

4. Download and Install following extentions for Editor

- Live Server : It is required to run web applications.
- vs-code-icons : It is required for user friendly icons.
- IntelliSense for CSS class names in HTML

5. Download and Install TypeScript on your PC

```
C:\> npm install -g typescript
C:\> tsc  -v
```

CREATING A TYPESCRIPT PROJECT

1. Create a new folder on your PC for typescript project

```
E:\typescript-project
```

2. Open the folder in Visual Studio Code

3. Open Terminal

Terminal Menu => New Terminal[Ctrl + `]

Change your terminal from Power Shell to Command Prompt

4. Run the following command in terminal

```
E:\>npm init -y
```

This will generate package. json, which comprises of project meta data.

- App Name
- Version
- Copyright
- License
- Dependencies
- Author etc...

5. Generate "tsconfig.json", it is required to set rules for typescript in project.

```
E:\>tsc  -init
```

1. Node JS 2. Editor VS Code

3. TypeScript 4. Create Project

Project File System:

- package.json : Comprises of project meta data.
- tsconfig.json : Comprises of typescript configuration settings.
- public : Comprises of static resources: images, html, text..
- src : Comprises of dynamic resource: js, ts, css, scss..
- index.html : It is the startup page for project.

Note:

If you face any problem like “running scripts disables” then type below three commands

```
set-ExecutionPolicy RemoteSigned -Scope CurrentUser
Get-ExecutionPolicy
Get-ExecutionPolicy -list
```

TYPESCRIPT LANGUAGE BASICS

1.Variables 2.Data Types 3.Operators 4.Statements 5.Functions

Note:

- Every TypeScript file will have extention ".ts"
- Transcompile the TypeScript file into JavaScript

```
                index.ts
  >tsc index.ts
```

- It generates "index.js"
- If your code comprises of presentation related to DOM then link "index.js" to any HTML page.

```
<script src="index.js"> </script>
```

- If your code comprises of console methods without any DOM then you can directly run JavaScript program using node compiler.

```
>node index.js
```

EX:

1. Go to "Src" folder
2. Add a new file

```
    "index.ts"
var username:string = "John";
document.write("Hello ! " + username);
```

3. Right click on "index.ts" and select "Open in Intergrated Terminal"

4. Transcompile
> tsc index.ts [generates index.js]

5. Go to "index.html"

```
<head>
   <script src="src/index.js"> </script>
</head>
```

Test from Console:

1. Go to index.ts

```
var username:string  = "John";
console.log("Hello ! " + username);
```

2. Transcompile

```
> tsc index.ts
```

3. Run using Node Compiler

```
> node index.js
```

VARIABLES

- Variables are storage locations in memory, where you can store a value and use it as a part of any expression.

- Variable configuration comprises of 3 phases

a) Declaration:
b) Assignment
c) Initialization

```
var x;          => Declaring
x = 10;         => Assignment
var y = 20;     => Initialization
```

- TypeScript variables are declared by using the following keywords
  a) var
  b) let
  c) const

a) Var :

- It defines a function scope variable.
- You can declare in any block of a function and access from any another block.
- It allows declaring, assignment and initialization.

Ex:

```
function Demo()
{
   var x;
   x = 10;
   if(x==10)
   {
```

```
        var y = 20;
    }
    console.log("x=" + x + "\n" + "y=" + y); // 10, 20
}
Demo();
```

- It allows shadowing.
- Shadowing is the process of re-declaring or re-initialization of any variable within the scope.

Ex:

```
function Demo()
{
    var x;
    x = 10;
    if(x==10)
    {
        var y = 20;
        var y = 30;     //shadowing
    }
    console.log("x=" + x + "\n" + "y=" + y); // 10 30
}
Demo();
```

- It allows hoisting. Hoisting means we can use variables before declaring it.
- Hoisting allows the compiler to access variable declaration from any location. There is no order for declaring and using.

Ex:

```
    x = 10;
    console.log("x=" + x);
    var x;                      // hoisting
```

a) Let :

- It configure block scope variable.
- It is accessible in the block where it is defined and also to its inner blocks.
- It allows declaring, assignment, initialization
- It will not allow shadowing and hoisting.

Ex:

```
function Demo()
{
    let x;
    x = 10;
    if(x==10)
    {
        let y = 20;
        console.log("x=" + x + " y=" + y);
    }
```

```
}
Demo();
```

a) Const :

- It is also block scoped.
- It allows only initialization. [no declaring and assigning]
- It will not allow shadowing and hoisting.

Syntax:

```
const x;          // invalid
x = 10;           // invalid

const x = 10;     // valid
x = 20;           // invalid
```

TYPESCRIPT DATA TYPES

- Data Type defines the data structure.
- It defines the type, range and behaviour.
- TypeScript is a strongly typed language.
- It can set restriction for data reference.
- TypeScript data types are same as JavaScript types classified into 2 groups

a) Primitive Type
b) Non Primitive

Syntax:

```
var  variableName:dataType = value;
```

: Inheritance operator . Invoking operator

Note: If data type is not defined for variable in typescript then the default type is "any".

```
var  variableName;      =>     variableName:any
```

TypeScript supports "Type Inference", the data type will be determined according to value initialized.

```
var x;              x:any
var x = 10;         x:number
```

Type Inference is based on the value initialized not assigned.

```
var x;              x:any
x = 10;             x:any
x = "John";         x:any
```

Primitive Types:

- They are immutable types.
- Have fixed range for values.
- Structure can't change
- Stored in memory Stack [LIFO]

a)number b)string c)boolean d)null e)undefined f)symbol

Number

- It can handle

signed integer	- 45
unsigned integer	45
floating point	34.45
double	345.56
decimal	3450.56
binary	0b101
hexa	0 to f
octa	0o495
bigint	Binary [format]
exponent	2e3 2 x 10[3] = 2000

Syntax:

```
var price:number = 3400.56;
var rate:number = 2e3;            [2000]
var bit:number = 0b1010;          [10]
```

Ex:

```
const bit:number = 0b1010;
const exp:number = 2e3;
console.log("Bit=" + bit + "\n" + "Exponent=" + exp);
```

- To verify the input type number or not, we use the function "isNaN()".
- To convert string format numeric value into number we use
  a) parseInt()
  b) parseFloat()

```
<script>
    document.write(Number.MIN_SAFE_INTEGER);
     document.write(Number.MAX_SAFE_INTEGER);
</script>
```

String Type

- String is a literal with group of characters enclosed in
  a) Single Quote ' '
  b) Double Quote " "
  c) BackTick ` `
- Back Tick is new from ES5+, it allows embedded expression "${ }"

Syntax:

```
var link:string = "<a href='home.html'>Home</a>";
var link:string = '<a href="home.html">Home</a>';
```

- Single and double quote uses "+" to concat expression.
- Back tick uses embedded data binding expression "${}"

Ex:

```
var username:string = "John";
var age:number = 23;
var msg1:string = "Hello !" + " " + username + " " + "you will be" + " " + (age+1) + " "
+ "next year";
var msg2: string = `Hello! ${username} you will be ${age+1} next year`;
console.log(msg1);
console.log(msg2);
```

- String formatting and manipulation methods are same in TypeScript.

String Formatting Methods:

bold(), italics(), sup(), sub(), fontsize(), fontcolor(), toUpperCase(), toLowerCase()

String Manipulations Methods:

indexOf(), lastIndexOf(), charAt(), charCodeAt(), startsWith(), endsWith(), slice(), substr(), substring(), match(), split(), trim(), length etc..

Ex:

```
var mobile:string = "+(44)(30) 2242 4563";
if(mobile.match(/\+\(44\)\([0-9]{2}\)\s[0-9]{4}\s[0-9]{4}/)) {
  console.log("OTP Sent");
} else {
 console.log("Invalid Mobile");
}
```

Boolean Type

- Boolean types are used in decision making.
- Boolean type can handle
  a) true
  b) false
- JavaScript boolean type "true = 1" and "false = 0".
- TypeScript will not allow 1 and 0 for boolean. You have to use "true or false".

Syntax:

```
var stock: boolean = true;
if(stock==1)    // invalid in typescript but valid in Java script
{ }
if(stock==true)// valid
{ }
```

Note: TypeScript supports "Union of Types".

```
var name:string|number;
 var username:string|null = prompt("Enter Name");
 var username:string = prompt("Enter Name");    // invalid
```

Null and Undefined

- Null specifies that there is no value provided into reference at run time.

Syntax:

```
var  username: string | null = prompt("Enter Name");
```

- Undefined specifies that there is no value provided into reference during compile time.

Syntax:

```
var  username: string;          // invalid
console.log(username);

var username: string| undefined = undefined;
```

Note: You can verify the value defined or undefined by using following techniques.

a) Check with undefined

```
if(Price==undefined)
{ }
```

b) Check with defined

```
if(Price)
{ }
```

Ex:

```
var Name: string = "Samsung TV";
var Price: number| undefined;
Price = 35000.44;
if(Price) {
   console.log(`Name=${Name}\n Price=${Price}`);
}
else {
   console.log(`Name=${Name}`);
}
```

NON-PRIMITIVE TYPES

- They are mutable type.
- Their structure can change dynamically.
- No fixed range for value.
- Value range varies according to memory available.
- Stored in memory Heap.
- TypeScript Non-Primitive types

  Array 2. Object 3. Map

ARRAY TYPE

- Arrays are used to reduce overhead and complexity.
- Overhead means there is space & it is not used.

- Arrays were introduces into computer programming to reduce overhead by storing values in sequential order.
- Arrays can reduce complexity by storing multiple values under one name.
- Array can handle various types of values.
- Array can change its size dynamically.

Declaring Array:

```
var  name :string[];              // string type
var  name :number[];            // number type
var  name :any[];               // any type
var  name :string[] | number[] ;
```

Initialize or Assign memory for Array:

- You can initialize or assign memory for array by using 2 techniques

a) Array Meta Character "[ ]" b) Array Constructor "Array()"

Syntax: Meta Character

```
var  collection :string[] = [];       // declaring and initialization of memory
                    (or)
var collection :string[];
collection = [ ];                 // declaring and assigning of memory
```

Syntax: Array Constructor

```
var  collection :string[] = new Array();
                (or)
var collection: string[];
collection = new Array();
```

FAQ: What is difference between array [ ] and Array() ?

Ans: Array() constructor will not allow to initialize different types of values even when the type is "any".
Array() constructor is only for initialization of similar type of values.
The data type is defined based on the first value initialized into memory.
Array() will allow assignment of values for various types, not initialization.

Array meta character "[]" will allow to initialize or assign various types if type is "any".

FAQ: What is a Tuple?

Ans : It is a collection that can initialize or assign various types of values.

Syntax:

```
var  collection: any[] = [];          ==> Tuple
```

FAQ: What is array de-structuring?

Ans : It is the process of extracting elements from array and storing in individual references.

Syntax:

```
var  collection: any[] = [1, "TV", true];

var [Id, Name, Stock] = collection;
```

Ex:

```
var collection: any[] = [1, "TV", true];

var [Id, Name, Stock] = collection;

console.log(`Id=${Id}\n Name=${Name}\n Stock=${Stock}`);
```

FAQ: What type of values we can store Array?

Ans : Array can handle any type of value

a) Primitive b) Non Primitive c) Function

Ex:

```
var collection: any[] = [1, "TV", true, ["Delhi", "HYD"], function(){console.log("Function in Array")}];

console.log(collection[3][1]);
collection[4]();
```

Array Methods

Reading Values:

toString(), join(), slice(), find(), filter(), map(),

for loops

for interators [in, of]

Ex:

```
var categories: string[] = ["Electronics", "Footwear", "Fashion"];

for(var property in categories)
{
   console.log(`[${property}]-${categories[property]}`);
}
```

Adding Elements into Array:

push(), unshift(),splice()

Removing Elements from Array:

pop(), shift(), splice()

Sorting Elements

sort(), reverse()

Note: Array supports union of types. But you can't initialized union types, you have to assign.

Syntax:

```
var collection: string[]|number[] = [];
collection[0] = 10;
collection[1] = "A";
```

Note: Read-only prevents the changing of array

Ex:

```
var arr: readonly any[]=[12,"Jilu"];
arr. push(20); // Error because read-only prevents the changing of array
console.log (Array. toString(arr));
```

PDF-1 (Array Manipulations)

OBJECT TYPE

- Object is used to keep all related data and logic together.
- "Alan Kay" introduced concept of object into computer programming in early 1960's.
- OOP started in early 1967 with SIMULA 67.

Syntax:

```
let  tv  = {
   Key: value,
   Key: value,
   Key: function() { }
}
```

- Above syntax will not have any restriction for keys and value.
- TypeScript can set restriction for key and value.
- TypeScript object type is defined using "{ }"

Syntax:

```
let   tv: {Name: string, Price :number} = {
   Name: "Samsung TV",
   Price : 45000.44
}
```

- Every property defined for object is mandatory to implement.
- You can configure nullable property by using "?" [null reference character]

Syntax:

```
let  tv : {Name: string, Price: number, Stock?:boolean}  = { }
```

- You can define multiple optional properties.

Ex:

```
let tv:{Name: string, Price: number, Stock?:boolean} = {
   Name : "Samsung TV",
   Price: 45000.44,
```

```
}
console.log(`Name=${tv.Name}\nPrice=${tv.Price}`);
```

- If your object comprises of only data then it is known as JSON.
  [JavaScript Object Notation]
- You can access object related properties within object by using "this" keyword.
- You can access object properties outside object by using object name.
- Object related functionality is defined by using function inside object.

Syntax:

```
object : {
   Name: string,
   Total: any
 }
```

- You can configure any type of data in object

Syntax : Array

```
tv : { Name: string, Price: number, Cities: string[] }  =  {  }
```

Syntax : Embedded Object

```
tv : {Name: string, Price: number, Rating:{Rate: number, Count: number}} = { }
```

Ex:

```
let tv:{Name: string, Price :number, Qty :number, Cities :string[], Rating:{Rate: number,
Count :number} , Total :any, Print?:any} = {
   Name: "Samsung TV",
   Price: 45000.44,
   Qty: 2,
   Cities: ["Delhi", "Hyd"],
   Rating: {Rate:4.3, Count:600},
   Total: function(){
      return this. Qty * this. Price;
   },
   Print: function(){
      console.log(`
        Name=${this. Name}\n
        Price=${this. Price}\n
        Qty=${this. Qty}\n
        Total=${this. Total()}\n
        Cities=${this. Cities. toString()}\n
        Rating=${this .Rating. Rate} [${this. Rating. Count}]
      `);
   }
}
tv. Print();
```

Note: Object properties are string type and value can be any type.

```
let  tv : { "Name": string, "Price": number } = {  }
```

Array of Objects

Data Type : Object Type Array => { }[]
Value Type : Array of Objects => [{ }, { }]

Ex:

```
let students:{Name :string, Age :number}[] = [
   {"Name": "John", "Age": 23},
   {"Name": "David", "Age": 45}
];
for(var student of students) {
   console.log(`${student. Name} - ${student .Age}`);    }
```

(or)

```
let students: any[] = [ { }, { } ];
```

MAP TYPE

- It is same like object with and key and value.
- Key & value can be any type.
- In object keys are only string type only
- Map is faster that object
- It have implicit methods
  set(), get(), has(), clear(), delete(), values(), keys(), entries()

Syntax:

```
let   data: Map<any, any> = new Map();
data. set(1, " ");
data. set("", 0);
data. keys()
data. values()
```

FAQ: What is difference between object and map?

Ans : Object	Map
* Key and Value	Key and Value
* Key is string	Key can be any
* Requires explicit iterators &	It have implicit iterators
* Slow	Fast
* Structured	Schema less

DATE TYPE

- TypeScript "Date" type is used to configure and handle date and time values.
- All JavaScript date functions are same in typescript

Syntax:

```
let  Mfd: Date = new Date("year-month-day hrs: min: sec. milliSec");
```

getHours()	getDay()	toLocaleDateString()
getMinutes()	getDate()	toLocaleTimeString()

getSeconds() getMonth() setHours()
getMilliSeconds() getFullYear() setDate() etc...

Ex:

```
let Mfd:Date = new Date("2023-01-02 10:20:32.89");
console.log("Mdf=" + Mfd. toLocaleDateString());
```

SYMBOL

- It is a primitive data type of JavaScript introduced with E6. [ECMA Script 2016]
- It is used to configure a unique reference for object.
- It is hidden in iterations but accessible individually.

Syntax:

```
<script>
  let ProductId = Symbol ();

  let product = {
    [ProductId] : 1,
    Name: "TV",
    Price: 40000.44
  }

 for(var property in  product)
 {
  console.log(product[property]);
 }
  console.log("ProductId=" + product[ProductId]);
</script>
```

Ex:

```
let ProductId = Symbol();

let product: any = {
   [ProductId]: 1,
   Name : "TV",
   Price: 45000.44
};
for(var property in product)
{
   console.log(property);
}
console.log("ProductId=" + product[ProductId]);
```

OPERATORS

- TypeScript Operators same as JavaScript

1. Unary ++, -- 2. Binary +, -, * 3. Ternary ?:

1. Arithmetic 2. Logical 3. Assignment 4. Comparison 5.Bitwise 6. Special

- New Operator from ES5 "**" [Exponent]

```
2**3            = 8     ES5
Math. pow(2,3)  = 8     ES4
```

Q What is difference between "==" & "===" ?

```
==          can compare values of different types
===         can compare values only of same type.
10=="10"    true
10==="10"   false
```

TYPESCRIPT STATEMENTS

1. Selection Statements
   if, else, switch, case, default

2. Looping Control Statements
   for, while, do while

3. Iteration Statements
   for.. in, for.. of

4. Jump Statements
   break, continue, return

5. Exception Handling Statements
   try, catch, throw, finally

TYPESCRIPT FUNCTIONS

- A function is used in "Refactoring" the code.
- Refactoring is a mechanism of encapsulating a set of statements and extracting to a function or file.

Syntax:

```
function  Name(params)
{
    statements;
}

function  Name(params)   => Declaration
Name(params)             => Signature
```

{ } => Definition

- Function Parameters - Anonymous Function - Function Recursion - Arrow Function
- Function Recursion - Function Return - Function Closure

Note: TypeScript functions are same as JavaScript but they are configured with data type for both function and parameters.

Syntax:

```
function Name (param: type): type
{
}
```

Ex:

```
function Hello(username: string):string
{
   return `Hello ! ${username}`;
}
function Welcome():void
{
   console.log("Welcome to TypeScript");
}
console.log(Hello(" john"));
Welcome();
```

- If function is not returning any value then it is configured as "void".

```
function  welcome():void
{ }
function  addition(a: number, b:number) : number
{
 return a + b;
}
```

- You can define optional parameters in TypeScript function.

```
function Details(Name: string, Price?:number)
{ }
```

- You can't define a required parameter after optional parameter.
- Optional parameters must be last parameters.

```
function Details(Name: string, Price? number, Stock: boolean)
{ }       // not valid
```

Ex:

```
function Details(Name: string, Price?:number):void {
   if(Price)
   {
      console.log(`Name=${Name}\n Price=${Price}`);
   } else {
      console.log(`Name=${Name}`);
```

```
    }
}
Details("Samsung TV", 34000.33);
```

TYPESCRIPT OOPS

@ Real world application development uses 3 types of programming systems ;
- a) POPS [Process Oriented Programming System]
- b) OBPS [Object Based Programming System]
- c) OOPS [Object Oriented Programming System]

@ Real world application development uses various programming approaches ;
- a) Functional Programming
- b) Structural Programming
- c) Imperative Programming
- d) Procedure Oriented Programming etc..

POPS:

- It supports low level features.
- It can directly interact with hardware.
- It is faster.
- It uses less memory.

Ex: C, Pascal, COBOL

- Code reusability Issues
- Code separation Issues
- Dynamic memory issues
- Code security
- Extensibility

OBPS:

- It supports code reusability
- It support code separation
- It support dynamic memory
- It support limited extensibility

Ex: JavaScript, Visual Basic

- No dynamic polymorphism
- No code level security

OOPS Features	OOP Characteristics
- Code reusability	Inheritance
- Code separation	Aggregation
- Code extensibility	Polymorphism
- Dynamic polymorphism	Encapsulation
- Code level security	Abstraction

Ex: C++, Java, C#

- Can't interact with hardware directly
- Don't support low level

- Need more memory, Complex
- Heavy & Slow also

CONTRACTS

- Contract defines rules for designing any component.
- Contracts are designed by using "interface".
- Interface contains the rules for designing a component.

Syntax:

```
interface Name
{
   // rules
}
```

- Interface must contain only rules not any implementation.

```
interface  Name
{
  Property: Type = value;       // invalid
}
```

Ex:

```
interface   iProduct
{
  Name: string;
  Price: number;
  Stock: boolean;
}

let  product: iProduct = {
     Name: "Samsung TV",
     Price: 34000.44,
    Stock: true
}
```

- Contract can have optional rules
- Optional rules are defined by using "null reference character - ?"
- Optional rules are required to design goals.

Syntax:

```
interface iProduct
{
  Name: string;            ]
  Price: number;           ]  Objectives(These are mandatory to fulfil ]
  Stock: boolean;          ]
  Rating?:number;           => Goal(These are not mandatory to fulfil if we fulfil then better &
}                                    if it is not fulfilled then no problem)
```

- A contract can have read-only rules.
- Readonly rule will not allow to assign a value after initialization.

Syntax:

```
interface  IProduct
{
  Name: string;
  readonly Price: number;
}
let  product: IProduct = {
  Name: "TV",
  Price: 45000.44,
}
product. Name = "Samsung TV";       // valid
product. Price = 60000.44;              // invalid – readonly
```

- Contract can have rules for methods.
- Method rule comprises of declaration and signature.
- Method rule can't have definition.
- Method name in contract is defined by using "( )".

Syntax:

```
 interface IProduct
 {
   Name: string;
   Total():number;
   Print():void;
 }
```

- Method rule can define parameters.

Syntax:

```
 interface  IProduct
 {
    Details(Name: string, Price: number): void;
 }
```

Ex:

```
interface IProduct
{
    Name: string;
    Price: number;
    Qty:  number;
    Total():number;
    Print?():void;
}
let product: IProduct = {
    Name : "TV",
    Price: 45000.44,
    Qty: 2,
    Total: function(){
       return this. Qty * this. Price;
    },
    Print: function(){
```

```
        console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nTotal=${this.
Total()}`);
    }
}
product. Print();
```

- You can extend contracts.
- You have to extend contracts in order to achieve "Backward Compatibility".
- Backward Compatibility allow users to use older version and its features even when a new version is available.

Syntax:

```
interface  version1
{
   A:string;
   B:string;
}
interface  version2 extends  version1
{
  C:string;
}
```

Ex:

```
interface Hdfc_Version1_Contract
{
   Personal: string;
   NRI: string;
}
interface Hdfc_Version2_Contract extends Hdfc_Version1_Contract
{
   Loans: string;
}
let low_compatible_user:Hdfc_Version1_Contract = {
   Personal : "Personal Banking",
   NRI: "NRI Banking",
}
let high_compatible_user:Hdfc_Version2_Contract = {
   Personal : "Personal Banking",
   NRI : "NRI Banking",
   Loans: "Personal and Housing Loans"       }
```

- Contract support multiple and multilevel extentions. [Inheritance]

FAQ: What is backward compatibility?

FAQ: What is side by side execution?

ANS : It is the ability of running multiple versions of same application on a device.

Ex:

```
interface Hdfc_Version1_Contract
{
   Personal: string;
```

```
    NRI: string;
}
interface Hdfc_Version2_Contract extends Hdfc_Version1_Contract
{
    Loans: string;
}
interface Hdfc_Version3_Contract extends Hdfc_Version2_Contract
{
    GovtSchemes: string;
}

let low_compatible_user:Hdfc_Version1_Contract = {
    Personal : "Personal Banking",
    NRI: "NRI Banking",
}
let high_compatible_user:Hdfc_Version2_Contract = {
    Personal : "Personal Banking",
    NRI : "NRI Banking",
    Loans: "Personal and Housing Loans"
}
```

Ex: Multiple

```
interface IProduct
{
    Name: string;
    Price: number;
}
interface ICategory
{
    CategoryName: string;
}
interface ProductContract extends IProduct, ICategory
{
    Title: string;
}
let obj :ProductContract = {
    Name : "TV",
    Price : 45000.44,
    CategoryName :  "Electronics",
    Title: "Samsung"
}
```

CLASS IN OOP

- Class is a program template.
- It comprises data and logic, which you can implement and customize according to requirements.
- Class have the behaviour of Entity or Model.
- If a class is mapping to business requirements then it is referred as "Entity".
- If a class is mapping to data requirements then it is referred as "Model".
- Class is used as a blue-print for creating of objects.
- Every class comprises of 2 types of configuration techniques
  a)Class Declaration b) Class Expression

a)Class Declaration & Expression:

- Class Declaration comprises of a constant set of members

Syntax:

```
class  Product
{
   // members
}
```

- Class Expression comprises of members, which can change according to state and situation.

Syntax:

```
 let  demo = class {
    // members;
  }
```

Ex:

```
var CategoryName: string = "Electronics";

var Demo = class {};

if(CategoryName=="Electronics"){
   Demo = class {
      ProductName = "TV";
      Price = 34000;
   }
} else {
   Demo = class {
      EmpName = "John";
      Salary = 35000;
   }
}
```

Class Members:

- Every class can contain only 4 types of members [Class Members]
  a) Property
  b) Method
  c) Accessor
  d) Constructor

FAQ: Can we declare a variable as class member?
Ans: No.

FAQ: Why variable is not allowed as class member?
Ans : Variables are immutable. Class can't have immutable members.

FAQ: Can we have a variable in class?
Ans: Yes. As a member of any method.

FAQ: Can we declare a function as class member?
Ans: No. Functions are immutable.

FAQ: Can we have a function in class?
Ans: Yes. As a member of any method.

Static Members in Class:

- A class can have static and non-static members.
- TypeScript class can have
    a) Static Property
    b) Static Method

Static:

- It refers to continuous memory.
- The memory allocated for first object will continue to next.
- Static is used to manage continuous operations.
- Static members are declared by using "static" keyword

```
static Property = "value;
static method() { }
```

-Static members are accessed inside or outside class by using class name.
-Static uses more memory and leads to memory leaks.

Non-Static | Dynamic:

- It refers to discreet memory.
- It is disconnected memory.
- Memory allocated for an object will be destroyed after object finished using class.
- It is safe but is not good for continuous operations.
- Non Static members are accessed with in the class by using "this" keyword and outside class by using instance of class.

Ex:

```
class Demo
{
   static s = 0;
   n = 0;
   constructor (){
      Demo. s = Demo. s + 1;
      this. n = this. n + 1;
  }
   Print () {
      console.log(`s=${Demo. s} n=${this. n} `);
   }
}
let obj1 = new Demo();
obj1.Print();

let obj2 = new Demo();
obj2.Print();

let obj3 = new Demo();
obj3.Print();
```

ACCESS MODIFIERS

- They define the accessibility of any member in a class.
- TypeScript supports following access modifiers

a) public b) private c) protected

- public is accessible from any location.
- private is accessible only with in the specified class.
- protected is accessible in derived class only by using derived class object.

Note: All members are accessible within the specified class.

PROPERTY

- Properties are used to store data.

Syntax:

```
public  Name: string = "John";
private Price: number = 45000.44;
protected Stock: boolean = true;
```

- You can store any type of data in property
  a) Primitive Type
  b) Non Primitive Type

Syntax:

```
public  Cities: string[] = [ ];
public  Rating:{Rate: number, Count: number} = { };
                (or)
public  Rating: IProduct = {  };
```

- Properties are mutable.
- Their state can change dynamically by using "Accessors"

Accessors:

- Accessor is used to give a fine grained control over any property.
- Accessors are 2 types
  a) Getter
  b) Setter
- Getter is used to read and return value.
- Setter is used to input and write value.
- You can read and write into a property by using accessors.

Syntax:

```
get  aliasName()
{
  return value;
}
set  aliasName(newValue)
{
   property = newValue;
}
```

Note: Accessors are available from JavaScript ES5 version

Ex:

```
var username: string | null = prompt("Enter Name");
```

```
var role: string | null = prompt("Enter Your Role" , "admin | manager | customer");
var productname: string | null = prompt("Enter Product Name");

class Product
{
   public _productName: string | null = null;

   get ProductName(){
     return this._productName;
   }
   set ProductName(newName: string | null){
      if(role=="admin"){
         this._productName = newName;
      } else {
         document. write(`Hello ! ${username} your role ${role} is not authorized to set
product name.`);
      }
   }
}

let tv = new Product();
tv. ProductName = productname;
if(tv. ProductName){
  document. write ("Product Name : " + tv. ProductName);
}
```

- You can also use accessors to access any property from multilevel hierarchy.

```
class Product
{
   public Name: string = "Samsung TV";
   public Rating: any = {
      CustomerRating: {Rate:3.4, Count:4600},
      VendorRating: {Rate:4.5, Count:300}
 }
   get VendorRating(){
      return this. Rating. VendorRating. Rate;
  }
}
let tv = new Product();
console.log(`Vendor Rating: ${tv. VendorRating}`);
```

Methods:

- All about methods is same as in JavaScript.
- You have to configure methods with access modifier and return type.

Syntax:

```
      public Total():number
      {
        return  0;
      }
```

```
    public Print():void
    { }
```
- Method is used for refactoring the code.
- Methods are mutable.

Constructor:
- Constructor configuration is same as in JavaScript
- Constructor is used for instantiation.
- Constructor is responsible for creating object for class.
- Constructor is a software design pattern.
- It is under "Creational Patterns" category.
- Constructor is anonymous.

Syntax:
```
    class Product
    {
      constructor() {
     } }
```
- It's parameters are same as method parameters.
- Constructor is a special type of sub-routine that executes automatically for every object.

Singleton: we can say a class as singleton if it has only one object.

FAQ: If constructor is parameterized when to pass values into constructor?
Ans: At the time of allocating memory for object.
```
    new className(values);
```

JavaScript Special Operators
- new
- void
- delete
- typeof
- instanceof
- in
- of etc..

Ex:
```
class Database
{
   constructor(dbName: string){
      console.log(`Connected with ${dbName} Database`);
   }
   public Insert():void{
      console.log("Record Inserted");
   }
   public Delete():void {
      console.log("Record Deleted");
   }
}
let ins = new Database("oracle");
ins. Insert();
```

- In TypeScript constructor can't overload.
- It will not support static and private constructor.

Class Implementation:

- Every class is designed as per the contract.
- Contract is implemented by class.
- A class can implement multiple contracts.

Syntax:

```
 class   className implements Contract1, Contract2
 {
   }
```

FAQ: Can we define any member additionally in a class which is not configured in contract?

Ans: Yes. [Class is a Template]

Ex:

```
interface IProduct
{
   Name: string;
   Price: number;
   Qty: number;
   Total():number;
   Print():void;
}
interface ICategory
{
   CategoryName :string;
}
class Product implements IProduct, ICategory
{
   public Name: string = "";
   public Price: number = 0;
   public Qty: number = 0;
   public CategoryName: string = "";
   public Stock:boolean = true;
   public Total(): number {
      return this. Qty * this. Price;
   }
   public Print(): void {
      console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nTotal=${t
his.Total()}\nCategory=${this.CategoryName}\nStock=${this.Stock}`);
   }
}
let tv  = new Product();
tv. Name = "Samsung TV";
tv. Price = 45000.44;
tv. Qty = 2;
tv. CategoryName = "Electronics";
tv. Print();
```

Class Inheritance:

- A class can extend another class.
- It supports single and multilevel inheritance.
- Class can't extend multiple classes. [Constructor Deadlock]
- TypeScript derived class constructor must call super class constructor.

Syntax:

```
class Super
{
 constructor(param?){ }
}
class Derived extends Super
{
  constructor(){
     super(value);
  }
}
```

Ex:

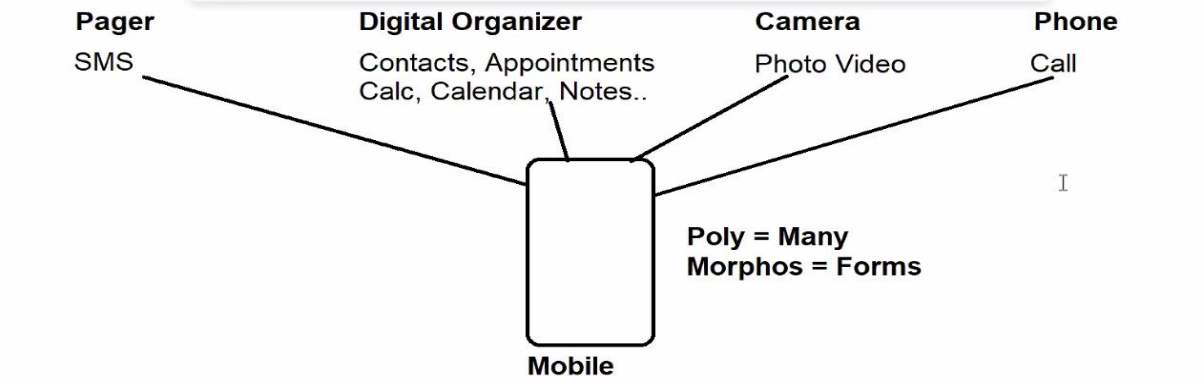
```
class Super
{
   constructor(){
      console.log("Super Class Constructor");
   }
}
class Derived extends Super
{
   constructor(){
      super();
      console.log("Derived class Constructor");
   }
}
let obj = new Derived();
```

Abstract class is an incomplete class i.e it contains the members that are incomplete so we cannot create an object for an abstract class.

POLYMORPHISM

- Poly means Many, Morphos means Forms.
- The ability of any component to serve for different situations is known as polymorphism.
- A component can have multiple behaviours.
- A component can change its behaviour according to state and situation.
- Instead of creating lot of components, you can create one component that exhibits polymorphism.
- Polymorphism is the process of overloading the memory with different functionalities.
- A method or function can overload.
- A class memory can overload.
- An object memory can overload.

- Technically polymorphism is a single base class reference can use the memory of multiple derived classes.




```
let employees: Employee[] = [new Admin(), new Developer().. ];
```

Ex:

1. index. ts

```
class Employee
{
   public FirstName: string = "";
   public LastName: string = "";
   public Designation: string = "";
   public Print():void {
      document. write(`${this. FirstName} ${this. LastName} - ${this. Designation}<br>`);
   }
}
class Developer extends Employee
{
   FirstName = "Raj";
   LastName = "Kumar";
   Designation = "Developer";
   Role = "Developer Role : Build, Debug, Test, Deploy";
   Print(){
      super. Print();
      document. write(this. Role);
   }
}
class Admin extends Employee
{
   FirstName = "Kiran";
   LastName = "Kumar";
   Designation = "Admin";
   Role = "Admin Role : Authorizations and Authentication";
   Print(){
      super. Print();
      document. write(this. Role);
   }
}
class Manager extends Employee
{
   FirstName = "Tom";
```

```
    LastName = "Hanks";
    Designation = "Manager";
    Role = "Manager Role : Approvals";
    Print(){
        super. Print ();
        document. write (this. Role);
    }
}

let employees: Employee[] = [new Developer(), new Admin(), new Manager()];
```

2. index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src="src/index.js"></script>
    <script>
        var designation = prompt("Enter Designation");
        for(var employee of employees)
        {
            if(employee. Designation==designation) {
                employee. Print();
            }
        }
    </script>
</head>
<body>

</body>
</html>
```

TEMPLATES IN OOP

- Abstract classes are used to design templates.
- A template comprises of data and logic which is already implemented partially, so that developer and extend and complete the implementation according to client requirements.
- Abstract class contains both methods that are implemented and not implemented.
- Developer have to implement and customize the methods that are not implemented.
- If any method is incomplete then it is marked as "abstract".
- If any one member of a class is abstract then class is marked as "abstract".
- Abstraction is the process of hiding the structure of component and providing only the functionality.

Ex:

```
interface ProductContract
{
    Name: string;
```

```
    Price: number;
    Qty: number;
    Total():number;
    Print():void;
}
abstract class ProductTemplate implements ProductContract
{
    public Name: string = "";
    public Price: number = 0;
    public Qty: number = 0;
    public abstract Total():number;
    public abstract Print(): void;
}
class ProductComponent extends ProductTemplate
{
    Name = "Samsung TV";
    Price = 45000.44;
    Qty = 2;
    Total(){
        return this. Qty * this. Price;
     }
    Print(){
        console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nTotal=${t
his.Total()}`);
     }
}

let tv= new ProductComponent();
tv. Print();
```

GENERICS

- Generic refers to type safe content with strongly typed nature.
- A generic type is open to handle any type until the value is provided.
- A generic type and make the component as strongly typed after the value is assigned.
- TypeScript can have various Generic members,
  a) Method can be generic
  b) Property can be generic
  c) Parameter can be generic
  d) Class can be generic

Syntax:

```
    public Print<T>(a: T, b:T)
    {
        console.log(`a=${a}\n b=${b}`);
    }

    Print<number>(10, 20);
    Print<string>("A", "B");
```

Ex-1:

```
interface IProduct
{
   Name: string;
   Price: number;
}
interface IEmployee
{
   FirstName: string;
   Designation: string;
   Salary: number;
}
class Service
{
   public GetData<T>(data: T){
      for(var property in data){
         console.log(`${property} : ${data[property]}`);
      }
   }
}
let obj = new Service();
console.log(`----Employee Details----`);
obj. GetData<IEmployee>({FirstName: "John", Designation: "Manager", Salary: 50000});
console.log(`----Product Details-----`);
obj. GetData<IProduct>({Name: "TV", Price: 45000});
```

Ex-2:

```
interface IOracle
{
   UserName: string;
   Password: string;
   Database: string;
}
interface IMySql
{
   Host: string;
   User: string;
   Pwd: string;
   Db: string;
}
interface IMongoDB
{
   Url:string;
}

class Database<T>
{
   public ConnectionString: T| null = null;
   public Connect():void{
      for(var property in this. ConnectionString) {
         console.log(`${property}: ${this. ConnectionString[property]}`);
      }
   }
```

```
}
console.log(`------Oracle Connection---------`);
let oracle = new Database<IOracle>();
oracle. ConnectionString = {
   UserName: "scott",
   Password: "tiger",
   Database: "studentsDb"
}
oracle. Connect();

console.log(`------MySql Connection---------`);
let mysql = new Database<IMySql>();
mysql.ConnectionString = {
   Host: "localhost",
   User: "root",
   Pwd: "12345",
   Db: "EmpDb"
}
mysql.Connect();

console.log(`------MongoDB Connection--------`);
let mongo = new Database<IMongoDB>();
mongo. ConnectionString = {
   Url: "mongodb://127.0.0.1:27017"
}
mongo. Connect();
```

ENUM

- Enum refers to Enumeration, which is a set of constants.
- Every constant must be initialized.
- Enum in TypeScript can have following constants
  a) string
  b) number
  c) expression
- Enum number constants can have auto implementation.
- Auto implementation is not supported for string and expression.

Syntax:

```
enum  Name
{
  ref = value
}
Name. ref
```

Ex-1:

```
enum ErrorCodes
{
   OK,
   Success=201,
   NotFound = 404,
   InternalError
```

```
}
console.log(`Status Code for Success : ${ErrorCodes. Success}`);
```

- Enum expression must be a numeric return value or string.
- Boolean expressions are not allowed.

Ex-2:

```
enum ErrorCodes
{
  A = 10,
  B = 20,
  C = A + B
}
console.log(`Addition=${ErrorCodes. C}`);
```

- Enum supports reverse mapping. It allows to access a key with reference of value.

Ex-3:

```
enum ErrorCodes
{
  NotFound = 404
}
console.log(`${ErrorCodes. NotFound}: ${ErrorCodes[404]}`);
```

MODULES

- Module is a set of variables, functions, classes, contracts, templates etc.
- Modules are used to build a library for application.
- JavaScript requires various module systems to handle modules.
- The popular module systems are
    - Common JS
    - Require JS
    - UMD [Universal Module Distribution]
    - AMD [Asynchronous Module Distribution] etc..

Note: You view or change the current module system in your project
go to "tsconfig.json"
NPM installs a module system called common js.

- Every TypeScript or JavaScript file is considered as a Module.
    ProductContract.ts => Module Name : "ProductContract"

- In a module every member is private at module level. It is not accessible outside module.
- If you want any member accessible outside module then you can mark it as "export".

Syntax:

```
export interface Name { }
export function Name() { }
export class Name {  }
export const Name = function() { }
export const Name = class {  }
```

- A module can have one "default" export.

Syntax:

```
export default interface Name {  }
```

Note: Only one member can be marked as default. There can't be multiple defaults in a module. If we want to automatically load members into memory then mark it as default.

- The members of any module can be accessed and used in another module by importing into context.

Syntax:

```
import   {MemberName}  from  "ModuleName";
import   {IProductContract} from  "ProductContract";
```

- To import any default member you don't need "{ }".

Syntax:

```
import  MemberName  from  "ModuleName";
```

- We can import both default and non-default members, but the default member can't be last.

Syntax:

```
import  Default_Member, {MemberName} from "ModuleName";
```

Ex:

1. Add following folders into "library" folder
   - contracts
   - templates
   - component

2. Add a new file into contracts
   ProductContract.ts

```
export interface IProductContract
{
   Name:string;
   Price:number;
   Qty:number;
   Total():number;
   Print():void;
}
export default interface ICategory
{
   CategoryName:string;
}
```

3. Go to Templates folder and add
   ProductTemplate.ts

```
import ICategory, {IProductContract} from "../contracts/ProductContract";

export abstract class ProductTemplate implements ICategory, IProductContract
{
   public Name: string = "";
   public Price: number = 0;
```

```
    public Qty: number = 0;
    public CategoryName: string = "";
    public abstract Total(): number;
    public abstract Print(): void;
}
```

4. Go to Component and add
ProductComponent.ts

```
import { ProductTemplate } from "../templates/ProductTemplate";

export class ProductComponent extends ProductTemplate
{
    Name = "Samsung TV";
    Price = 40000.33;
    Qty = 2;
    CategoryName = "Electronics";
    Total(){
        return this.Qty * this.Price;
    }
    Print(){
        console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nTotal=${t
his.Total()}`);      }}
```

5. Add a new folder "app" with a new file
Index.ts

```
import { ProductComponent } from "../library/components/ProductComponent";

let tv = new ProductComponent();
tv.Print();
```

6. Compile Index.ts

```
> tsc  index.ts
> node index.js
```

Ex: JavaScript Module

1. Demo.js

```
export function Hello(){
    return "Hello ! Welcome to Modules in JavaScript";
}
export function Addition(a,b) {
    return a + b;
}
```

2. Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script type="module">
        import {Hello, Addition} from "../library/demo.js";
        document.querySelector("p").innerHTML = Hello() + "<br>" + "Addition=" +
Addition(20,30);
    </script>
</head>
<body>
    <p></p>
</body>
</html>
```