Type Scripting

* It is handled by Microsoft .
* The typescript file has an extention .ts
* It creates a corresponding .js file for each .ts file
* It is Type safety.

# A normal type script function

function displayName(name:string){

console.log(name);

}

displayName("sabari");

# How to execute a type script??

* In the terminal make sure it is pointing to the folder you have created.
* Then execute the typescript file by,

PS D:\Bhuvana> tsc demo.ts

* Now the Javascript file will be generated automatically,
* Now you have to execute the .js file like,

PS D:\Bhuvana> node demo.js

# Type script with optional parameter

**syntax: lastName?:string**

function displayNamee(firstName:string,lastName?:string){

console.log(firstName+lastName);

}

displayNamee("sabari");

# Type script with default parameter

**syntax: lastName:string=””**

function displayNamee(firstName:string,lastName:string=""){

console.log(firstName+lastName);

}

# Looping

## Looping through the index

let fruits:string[]=["Apple","Orange","Banana","Kiwi"];

for(var c in fruits){

console.log(fruits[c]);

}

## Looping using the variable directly

let fruits:string[]=["Apple","Orange","Banana","Kiwi"];

for(var c of fruits){

console.log(c);

}

# Declaring variable in Typescripting

## using "var"

*Note: If you use var the variable name will be the same in the corresponding .js file also*

function call(name:string){

if(name=== "Bhuvana"){

var age =10;

console.log(age);

} var age =20;

console.log(age);}

call("Bhuvana")

**o/p = 10**

## using "let"

*Note: If you use var the variable name will be different in the corresponding .js file also*

function call(name:string){

if(name=== "Bhuvana"){

let age =10;

}

let age =20;

console.log(age);

}

call("Bhuvana")

**o/p = 10 , 20**

# tuple

*Note:It can have any datatype(number,string,boolean). It should contain only 3*

let employee:[number,string,boolean];

employee=[101,"Bhuvana",true]

for(var e of employee){

console.log(e);

}

# Custom Datatype

type custom = 12 | 23 |34;

let rollNumber:custom;

rollNumber =13;

\*\*\* Combination of datatypes \*\*\*

type custom = 12 | "Bhuvana" |34;

let rollNumber:custom;

rollNumber ="Bhuvana";

# Union Types

*Note: It can have any number of datatypes using pipe(|) operator*

function addTen(value:number|String|boolean){

console.log(value);

}

addTen("12");

## using parse int

function addTen(value:number|String){

if(typeof value === "string")

console.log(parseInt(value) + 10);

else

console.log(value + 10);

}

addTen("22");

# Functions

**Example 1:**

function add(a:number,b:number):number{

let x = a+b;

return x;

}

console.log(add(10,20));

**Example 2:**

let add2 = function(a:number,b:number):number{

let x = a+b;

return x;

}

console.log(add2(10,20));

# lambda function / Anonymous function

*Note: Here we dont have to use the "function" word*

let add3 = (a:number,b:number) => {

return a+b;

}

console.log(add3(10,20));

# Difference between normal and lambda function…

const inventory=[

{name :'apples',quantity:4},

{name :'berries',quantity:3},

{name :'oranges',quantity:8},

]

Note: here you have to declare a function for comparing the logic

**//normal function declaration**

function findberries(fruit){

if(fruit.name==="berries"){

// return fruit.name==='berries';

return 'berries is good for health';

}

}

let fruits = inventory.filter(findberries); **// without anonymous function**

**(or)**

let fruits = inventory.filter(f => f.name=== "oranges"); **// lambda function. here you don’t have to declare a function for comparing the logic**

console.log(fruits);

*Note: ‘Const’ and ‘let’ acts the same way*

# Expression Syntax

Syntax- ${}

Note: All the dynamic values should be given inside ``. For Example, ` The Score is : ${score}`

# Class

## Constructor

* Constructor is the keyword for generating the constructor
* Only one default constructor is allowed in typescript
* The default access specifier is public.
* This keyword is to point to the cuurent instance of the class.
* Local variable always require “let” declaration.

*Note : Classes in typescript are compiled as function in javascript*

## Default constructor :

constructor(){

}

## Creating an object:

let c1 = new Car();

## Declaring the member variable inside the constructor:

**Procedure 1: This is the standard way of declaring**

Public names:String; // member variable

Constructor(names:String){ //local variable inside the constructor

this.names = names; //now the local variable refers to the member variable

{

**Procedure 2: It is declared easier in typescript**

constructor(public names:String){

}

// here it is simplified. We can directly pass member variable inside the constructor

# Getters and setters :

**Standard way:**

public setNames(name:string){

this.name = name;

}

public getNames():string{

return this.name;

}

let c1 = new Car();

c1.setNames("Audi");

//console.log(c1.move());

console.log(c1.getNames());

**Typescript way:**

* \_ is an universal way of understanding that the developer is using getters and setters.
* “\_” is used as the member variable

private \_name: string;

public set names(name:string){

this.\_name = name; }

public get names():string{

return this.\_name;

}

c1.names = "Audi"; // now you can set the values directly using the method name.

console.log(c1.names); // also get the value using the same method name.

*Note: \_ is used in the member variable*

***\*\*\* Note:***

***If you get the error “Accessors are only available when targeting ECMAScript 5 and higher.”***

***While compiling give it as***

***PS D:\Bhuvana> tsc demo8.ts --target es5***

***PS D:\Bhuvana> node demo8.js***

***Program:***

class Car{

//member variable or class level variable

//public names:String;

private name:string;

private \_name: string;

//constructor(public names?:String){

//this.names="Default Name";

//}

constructor(){

}

public set names(name:string){

this.\_name = name;

}

public get names():string{

return this.\_name;

}

//member function

move():String{

let names:string = "Honda"; // local variable

return this.names +"Car Moves"; // class level variable

}

}

let c1 = new Car();

//c1.setNames("Audi");

//console.log(c1.move());

//console.log(c1.getNames());

c1.names = "Audi";

console.log(c1.names);

# Abstract Class:

* Both abstract method and concreate method cant create an instance
* Abstract Class acts as a parent class

## Defining an abstract class in TS:

This is same like java. The class name is preceeded with abstract keyword.

abstract class Machine{

}

## Abstract method:

abstract moreInfo():string;

**Note :**

Returning the member variable with dynamic value.

Standard way:

return this.manufacturer +"makes this machine"

TS way:

return `${this.manufacturer} makes this machine`

**Program:**

//Abstract Class acts as a parent class

export namespace mfg{

export abstract class Machine{

constructor(private manufacturer:string){

}

//concrete method

summary():string{

return `${this.manufacturer} makes this machine`

}

abstract moreInfo():string;

}

export class Cars extends Machine{

constructor(manufacturer:string){

super(manufacturer);

}

moreInfo():string{

return "Cars are safe";

}

}

let c4 = new mfg.Cars("Hyundai");

console.log(c4.summary());

console.log(c4.moreInfo());

}

# Importing and Exporting:

* Export the class like this,

export class Cars extends Machine{

}

* Import the exported class in another class like this,

import {Cars} from './demo9'

(or)

import \* as Bundle from './demo9'

class SelfDrivingCar extends Cars{

}

# Namespace:

Name space is used for just grouping the classes

While exporting a namespace, All the classes inside the namespace should also be export.

It should be the topmost line in the class.

## Exporting a Namespace:

export namespace mfg{

export abstract class Machine{

}

export class Cars extends Machine{

}

}

## Importing a namespace:

import {mfg} from './demo9'

class SelfDrivingCar extends mfg.Cars{

}

# Interface:

It is same like java Interface.

import {mfg} from './demo9'

//import \* as Bundle from './demo9'

class SelfDrivingCar extends mfg.Cars{

}

interface Iflyer{

fly():string;

}

interface Icruise extends Iflyer{

sail():string;

}

class Bird implements Iflyer{

fly():string{

return "Bird Flies with feathers"

}

}

let b = new Bird();

class Airplane implements Iflyer,Icruise{

sail(): string {

throw new Error("Method not implemented.");

}

fly(): string {

throw new Error("Method not implemented.");

}

}

https://github.com/SabariRepo/fullstackbatch6Sep2019