1. Installing + Setting up the editor.  
     
   npm install -g typescript  
   tsc .\usingts.ts  
   Extensions : ESLint : code quality check support.

Material icon theme.  
Path Intellisense : better support when we work with imports.  
Prettier - Code formatter  
debugger for chrometsc init

1. npm install --save-dev lite-server : server index.html file [something like live-server]
2. core types : javascript knows and typescript(lowercase) also supports:  
   number : +ve , -ve, floats  
   string : “abc”, `${template literals}`

boolean : true / false  
…objects and so on.

const add = (n1: number,n2: number) => {  
 return n1+n2;  
}  
const number1 = 5;  
const number2 = 2.8;  
add(number1,number2);

1. object types.  
     
   const person : {name:string;age:number} = {  
    name:"steve",  
    age:33  
   }  
   console.log(person.name)
2. Nested objects and types.  
     
   const person : {name:string;age:number;address:{ line1:string;line2:string;pin:number} } = {  
    name:"steve",  
    age:33  
   }
3. Array – can have arrays with strings and numbers mixed.

Types of the array can be flexible or strict.  
  
const person:{name:string;age:number;hobbies:string[]}= {

name:"steve",

age:33,

hobbies : ['sports','cooking']

}

for(const hoby of person.hobbies){

console.log(hoby.toUpperCase())

//typescript automatically detects that this is a string

}

person.hobbies.//map/pop/push/reduce... etc all will be listed automatically

1. Working with Tuples – only with TS  
   [1,2] – fixed length array + fixed type array  
     
   const person: { name:string; age:number; hobbies:string[]; role:[number,string] }= {   
   name:"steve",  
   age:33,  
   hobbies : ['sports','cooking'],  
   role:[2,'dev-eng'] // WHY ? here we wanted to have an array with exactly to elements  
   }

person.role.push(22);

person.role.push('admin'); // we only need two elements - but still TS does allow this - size can not controlled by typescript

// person.role = [1,'simple-role','another-fake-entry'] - when assigning it this way Ts complains about the third entry

person.role[1] = "test-eng"; // can switch values

// person.role[1] = 20; //this will be complained by typescript - the order of the types is strict.

**// So if you have scenario that there should be exactly only two elements in an array**

**// + And you know the type of each element in advance : TUPLE is the perfect solution**

1. Working with Enums (custom type in typescript): There might global constants – which are represented as numbers, but for which you want to assign a label.

**WITHOUT ENUMS : with constants**  
const ADMIN = 0;

const READ\_ONLY\_USER = 1;

const AUTHOR = 2;

const person = { name:"steve", role: READ\_ONLY\_USER }

if(person.role === READ\_ONLY\_USER){

console.log("The role is READ\_ONLY\_USER")

}

// The advantage of this is instead of using the numbers we use strings which are more readable

// The downside is ANY number can be stored + v have lot of constants and we have to manage them

role: 100  
  
**WITH ENUMS**enum Roles {

ADMIN,READ\_ONLY\_USER,AUTHOR

};

const person = { name:"steve", role: Roles.READ\_ONLY\_USER }

if(person.role === Roles.READ\_ONLY\_USER)

console.log("The role is READ\_ONLY\_USER")

1. **Union types(TS)**

const combine = (n1: number|string ,n2: number|string ) => {

if(typeof n1 === 'number' && typeof n2 === 'number')

{

return n1+n2;

}

else{

return n1.toString() + n2.toString();

}

}

console.log(combine(2,3))

console.log(combine('steve','jobs'))

//Thus we can use union types to be flexible regarding - what we do in a function.

1. **Literal types** : exact value it holds (based on core types) – used in conjunction with union types.  
     
   const combine = (n1: number|string ,n2: number|string, typeOfConversion : 'as-number' | 'as-string') => {

if(typeof n1 === 'number' && typeof n2 === 'number' || typeOfConversion === 'as-number')

{

return +n1 + +n2;

}

else{

return n1.toString() + n2.toString();

}

}

**console.log(combine('2','3','as-number')) // can’t use incorrectly.**

console.log(combine(2,'3','as-string'))

1. **Type Aliases / Custom Types.**

Why ? : No need to write “number|string” etc fully in multiple places.  
 : write code quicker with descriptive type aliases  
  
Example 1:   
**type combinable** = number | string;

**type conversionDescriptor** = 'as-number' | 'as-string';

const combine = (n1: **combinable** ,n2: **combinable**, typeOfConversion : **conversionDescriptor** ) => {

if(typeof n1 === 'number' && typeof n2 === 'number' || typeOfConversion === 'as-number')

{

return +n1 + +n2;

}

else{

return n1.toString() + n2.toString();

}

}

console.log(combine('2','3','as-number'))

console.log(combine(2,'3','as-string'))

Example 2 :   
**type User = {**

**name:string;**

**age:number**

**}**

const displayUser= (**emp : User**)=>{

console.log(emp.name + " is " + emp.age + " years old ")

}

displayUser(**{name:"steve",age:23}**)

1. Function return types and void(TS) + undefined.  
     
    const add = (n1:number,n2:number) : number=>{

return n1+n1;

}

//below does not return anything

const printResult = (n1:number) :**void**=>{

console.log(n1);

}

console.log(printResult(5));

//Above function returns undefined - this doesnot mean that we can return undefined explicitly

//and so the below call will fail.

// const printResult2 = (n1:number) : **undefined**=>{

// console.log(n1);

// }

//the point is you should void if a function returns nothing - this fn does not have a return statement.

const printResult3 = (n1:number) :**void**=>{

console.log(n1);

}

//then if you explicitly say that a fn returns undefined.

// - it means to typescript that - we have a return statement + we don't return a value.

const printResult4 = (n1:number) :**undefined**=>{

console.log(n1);

**return**;

}

//another valid case with return + void

const printResult5 = (n1:number) :**void**=>{

console.log(n1);

**return**;

}

//Note that : undefined is a valid type in javascript

const temp = **undefined**;

//the above statement will NOT throw any error.

1. Functions as types : describes the parameters and return values of the function  
   **Why ? :**   
     
   const add = (n1:number,n2:number) : number=>{

return n1+n2

}

const printResult = (n1:number) :void=>{

console.log('printing ' + n1)

}

let addRef:(a:number,b:number)=>number;

addRef= add;

//now typescript will complain

addRef = printResult;

//by adding "let addRef:Function;" below statements can be caught

//addRef = 5;

//But the point is we did not say which kind of function.

//when we say : addRef = printResult;

//we are storing the wrong function in there and typescript is not complaining about that

// we need too be presise about how the function should look like

console.log(addRef(5,2));

1. **Function types and callbacks.**//By mentioning void in the callback here - we are essentially saying we will ignore

// any result you might be returning in the callback you pass

// the below call back type does NOT force you to pass a 'callback that does not return anything'

// - but 'it says that it does not care whatever the callback you pass returns'

const addAndPrint = (n1:number,n2:number,cb:(n:number)=>void)=>{

cb(n1+n2);

}

addAndPrint(2,4,(nArg)=>{

console.log(nArg);

return "hurray";

//Typescript does not pick this error - even though void is mentioned in the parameter above

})

1. Unknown type.  
     
   let userInput : unknown ;

userInput = 44;

userInput = {

name:"11",

age:22

}

//so far : unknown is similar to any typescript type

let username:string = "Mac";

//username = userInput;

// the above will not be allowed because the type in userInput is not known.

if(typeof userInput === 'string'){

username = userInput;

}

1. **Never type** : another type functions can return.  
   Why ?  
     
   let userInput : unknown ;

userInput = 44;

userInput = {

name:"11",

age:22

}

//so far : unknown is similar to any typescript type

let username:string = "Mac";

//username = userInput;

// the above will not be allowed because the type in userInput is not known.

if(typeof userInput === 'string'){

username = userInput;

}

//This function never returns anything - not even undefined.

const generateError = (msg:string,errCode:number):never => {

throw {

message:msg,

errorCode:errCode

}

}

**----------------- TYPE SCRIPT COMPILER ------------------  
  
  
1.** Using watch mode. **>** tsc app.ts –w / --watch

**2.** If more than 1 ts files.

> Navigate to the folder where your typescript files are present.  
 > tsc –init  
 > tsconfig.json file is generated. Why ? It essentially tells typescript how it should compile the .ts files  
 > tsc  
 //the above command will convert all typescript files in the current folder to .js files.  
 // to run in watch mode for all files  
 > tsc -w / tsc –watch

3. How to include / exclude a file in the typescript compiler.  
 **exclude :** files you want to exclude from the ts compiler  
 **include :** Files you want to specifically include in the compilation process. Other files are excluded [basically we compile include minus exclude]  
 **files :** only individual files you want to compile, can not specify folders

{

"compilerOptions": {

/\* Basic Options \*/

/\* Strict Type-Checking Options \*/

/\* Module Resolution Options \*/

/\* Source Map Options \*/

/\* Experimental Options \*/

/\* Advanced Options \*/

},

"exclude": [ "node\_modules", "\*.analytics.ts"],

"include": ["\*.ts"],

"files": ["app.ts"]

}

4. ---------------- COMPILER OPTIONS-----------------------   
  
 **"target":** "es5",   
/\* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019' or 'ESNEXT'. \*/  
For which target javascript version we want to compile the code.  
The compiler also compiles the code that runs in certain set of browsers.  
ES6 : ES2015   
Alternatively we can use other transpilers.

**module**   
Why ? This options allows us to specify which default objects and features typescript knows.  
For example : Typescript should complain for the below statement , but it recognizes document.  
*const button = document.querySelector('button');*  
And the same will not work in the nodejs environment.  
This is controlled by the **lib** option. When uncommented ts shows a lot errors

**allowJs**   
Why ? we can allow javascript files in the compilation process.  
 **checkJs**Typescript compiler only check but does not compile

**.d.ts** : This matters if we are shiiping our project as a library.

**sourceMap**Why ? helps us with debugging and development.

In the developer tools window we only see the javascript files, This options helps us see the ts files there also and thus in debugging them too.  
  
**outDir and rootDir**

We can tell the typescript compiler where to look for all the typescript files and where to put the generated js files.  
The folder structure in the rootDir will be replicated in the outDir.

Let us say if you have two ts directories rootDir1and rootDir2 **:** Typescript willcreate two such directories in the outDir unless **rootDir** is specifically mentioned,which means to ignore other directories where ts files are present.  
 **noEmit :** The typescript compiler checks the files and reports any potential errors, but does not generate the javascript files.

**noEmitOnError:** If any ts file fails to compile the js files wont be generated.

5. ---------------- STRICT OPTIONS-----------------------   
 **noImplicitAny :** Thisensuresthe parameters type in mentioned, by default any will be taken and ts compiler will not allow this. This is not the case with variables ..for example you can say **let logged;** and later say **logged=true;**

Why ? functions are created first.  
 **strictNullChecks:** whenthis is false.The typescript compilerwill not complain in the below code.  
const button = document.querySelector('button'); [no need of the bang symbol]  
 **strictFunctionTypes** : Related to function types, not the types inside of functions. Bugs could be introduced if we work with classes and inheritance, we could introduce bugs.

**strictBindCallApply :** Usefulso that we don’t use bind call and apply accidentally in a way that does not work with our code.

**strictPropertyInitialization :** important when we work with classes.

**noImplicitThis :** has to do with the this keyword and TS basically tries to warn you when you use the this keyword in a place when it is not clear what it refers to.

5. ---------------- CODE QUALITY OPTIONS-----------------------   
  
 noUnusedLocals : we don’t want unused local variable  
 noUnusedParameters : we don’t want to keep unused parameters.  
 noImplicitReturns : All branches in this function must return.

6. ---------------- DEBUGGING -----------------------

Install : debugger for chrome + enable source maps in tsconfig.json.  
 put the breakpoint in the editor and hit start debugging.

7. <https://kangax.github.io/compat-table/es6/>  
Why ? which feature of Js is supported in which TS version and some browser related activity.

8. ---------------- NEXT GEN JAVASCRIPT -----------------------   
   
**LET AND CONST**  
 const userName = "steve"

//userName = "jobs"

//variables defined with the const keyword can not be changed.

let age = 29;

age=30;

// value can be changed - this is similar to var, but dont use var -

// Scope in which the variable is available.

// function foo(){

// var mat = "true"

// }

// console.log(mat);

// the above code will work in JS

//let and const introduced a new scope called block scope - avialable in the block they are defined and lower blocks

**ARROW FUNCTIONS (**ADDED IN ES6**):**

1 expression in the body of the function : const add = **(a,b) => return a+b;**  
1 param in JS : const printop = **op => console.log(op)**

1 param in TS : const printop = **(op : string | number) => console.log(op)**

//With the type assignment it will not work in typescript

//If we have the function type assigned to the constant.  
1 param in TS : const printop2: (ab :number | string ) =>void = **op => console.log(op)**

**DEFAULT FUNCTION PARAMETERS:**

They have to be last.  
const Increment= (a:number,b:number = 10) => { return a+b };

**THE SPREAD OPERATOR:** const hobbies = ['sports','cooking'];

// **add this array to other array using the spread operator**

const newHobbies = ["hiking",...hobbies]

newHobbies.push(...hobbies);

//console.log(newHobbies);

//**the spread operator also exists on objects**

const emp1 = {

name:'steve',

add : {

line1:'add-line1',

line2:'add-line2'

}

}

const emp2 = {...emp1}

console.log(emp2);

// we get a perfect copy of the original object and not just a pointer that points to the emp1 object.

**REST PARAMETERS:**  
 // we dont want to have fixed number of params - we want to allow as many values as the user   
 passes in.

// **A really useful feature for accepting an unlimited number of arguments.**

const add = (...numbers:number[])=>{

const matter = numbers.reduce((res,curValue)=>{ return res + curValue },10)

console.log(matter);

}

add(1,2,3);

**ARRAY AND OBJECT – DESTRUCTERING (pull elements out of the array):**

**WHY : PULLING OUT ELEMENTS FROM ARRAY AND OBJECTS.**

//array destructering - elements are pulled out in order

const hobbies:string[] = ['cooking','sports','stamp collection','painting']

const [firsthobby,secondhobby,...remaining] = hobbies;

console.log(firsthobby);

console.log(secondhobby);

console.log(...remaining);

//object destructering

const person = {

firstName : "steve",

age : 22,

address:{

line1:"add-line1",

line2:"add-line2"

}

}

//order does not matter

const {age, firstName,address:add} = person;

console.log(firstName);

console.log(add.line1);

---------------- CLASSES AND INTERFACES -----------------------

CONSTRUCTOR FUNCTIONS AND THE THIS KEYWORD.

class Department {

name:string;

//constructor method - reserved keyword

constructor(n:string){

this.name=n;

}

describe(this:Department){

//if ${name} is mentioned - the engine will look for such a property inside the describe method

// or outside of this class. - it will not search for the same in the class.

console.log(`This is an ${this.name} department` )

}

}

//new keyword - TS + JS : used to create the object

const accounting = new Department("Accounting")

accounting.describe()

//this keyword can be tricky

const accountingCopy = {

name:"Acccopy",

describe : accounting.describe

// we are passing the function itself to describe here.

// it does indeed execute the method here.

// Now, when the method is getting executed --

// -- 'this' in the console statement will not refer to the object the method was part of originally

// and typically 'this' will be refering to the object/thing which is responsible for calling a method.

// And in this case it is 'accountingcopy'.

// now 'accountingcopy' does not have a name property so .. it will say undefined.

//To work around the 'this' problem in TYPESCRIPT - we need to add a special parameter called 'this' to the describe function.

// This will catch unwanted behaviours

}

accountingCopy.describe();

PUBLIC AND PRIVATE ACCESS SPECIFIERS.

class Department {  
 name:string;

//access modifier - public is default access modifier

private employees : string[] = [];

//constructor method - reserved keyword

constructor(n:string){

this.name=n;

}

addEmployee(emp:string){

this.employees.push(emp);

}

}

//new keyword - TS + JS : used to create the object

const accounting = new Department("Accounting")

accounting.addEmployee('max')

accounting.addEmployee('min')

accounting.employees.push('ani'); // with private this line will not be allowed

// we dont want this to happen - we may need validation etc employees should not be accessed using 'private'

//thus employees is accessible from INSIDE the class only.

SHORT HAND INITIALIZATION.  
 class Department {

//private name:string; - not needed

// Creates properties for this class with the exact same names

constructor(public name:string){

//this.name = name; - not needed

}

describe(){

console.log(this.name)

}

}

const accounting = new Department("Accounting")

accounting.describe()

READ ONLY PROPERTIES. [only in TS]

If you don’t want some property to change after their initialization.  
   
class Department {

//private name:string; - not needed

// Create properties for this class with the exact same names

constructor(public **readonly** name:string){

//this.name = name; - not needed

}

describe(){

console.log(this.name)

}

changeName(){

this.name="Finance";

}

//not allowed – compile error

}

const accounting = new Department("Accounting")

accounting.describe()

INHERITANCE.  
 class Department{

private DepName : string = "DEFAULT\_DEPARTMENT";

private DepID : string = "DEFAULT\_ID";

private Employees : string[] = [];

constructor(depName:string,depId:string){

this.DepName=depName;

this.DepID=depId;

}

addEmployee(empName:string){

this.Employees.push(empName);

}

describeDepartment(){

console.log(`This is a ${this.DepName} department with ID : ${this.DepID}`)

}

listEmployees(){

this.Employees.map((value,index,array)=>{console.log(value)})

}

}

const finDep = new Department("DEPARTMENT",'0');

finDep.describeDepartment();

finDep.addEmployee("Steve");

finDep.addEmployee("Mike");

finDep.listEmployees();

//use case - 1

//we can only inherit from ONE class ONLY.

//class ITDepartment extends Department{

//}

//As long as we dont add a constructor for ITDepartment -

// - the base class constructor is automatically used during instantiation of sub-class

//const itDep = new ITDepartment("software");

//itDep.describeDepartment();

//use-case 2

class ITDepartment extends Department{

constructor(id:string,private developers:string[]){

//super has to be the first line

super("IT\_DEPARTMENT","id")

}

listDevelopers(){

this.developers.map((value,index,array)=>console.log(value))

}

}

const ITDep = new ITDepartment('11',["ag","bag"])

ITDep.describeDepartment();

ITDep.listDevelopers();

OVERRIDING PROPERTIES & THE ‘protected’ MODIFIER.  
  
 class Department{

private DepName : string = "DEFAULT\_DEPARTMENT";

private DepID : string = "DEFAULT\_ID";

//private admins : string[] = [];

protected admins : string[] = [];

constructor(depName:string,depId:string){

this.DepName=depName;

this.DepID=depId;

}

addAdmins(empName:string){

this.admins.push(empName);

}

describeDepartment(){

console.log(`This is a ${this.DepName} department with ID : ${this.DepID}`)

}

listEmployees(){

this.admins.map((value,index,array)=>{console.log(value)})

}

}

const finDep = new Department("DEPARTMENT",'0');

finDep.describeDepartment();

finDep.addAdmins("Steve");

finDep.listEmployees();

class ITDepartment extends Department{

constructor(id:string,private developers:string[]){

//super has to be the first line

super("IT\_DEPARTMENT","id")

}

listDevelopers(){

this.developers.map((value,index,array)=>console.log(value))

}

//over-riding behaviour

//If Admins is private in base class we can not this

//below from outside this - we shall try to access this base class protected variable - it wont even be listed

addAdmins(){

// some custom logic

this.admins.push("itadmin-keren");

}

}

const ITDep = new ITDepartment('11',["ag","bag"])

ITDep.describeDepartment();

ITDep.listDevelopers();

NOTE : TYPE ASSIGNMENTS AND ACCESS MODIFIERS ARE SPECIFIC TO TYPESCRIPT

Getters and Setters (both in js and ts).

class Department2{

private depEmployees:string[] = [];

constructor(private depName:string)

{ }

describeDepartment(){

console.log(`This is a ${this.depName} department`);

}

get departmentEmployees(){

return this.depEmployees;

}

set departmentEmployees(value:string[]){

this.depEmployees = [...this.depEmployees,...value]

}

listAllEMployees(){

this.depEmployees.map( (value,index,array)=>{ console.log(value) } )

}

}

const dep2 = new Department2("IT\_DEP");

dep2.departmentEmployees = ["emp1","emp2"]

dep2.listAllEMployees();

//NOTE : get and set functions can be accessed like properties from outside

// but the advantage is that we can add some extra logic in these

//NOTE THAT the setters and getters cannot have the same name as the private property

STATIC METHODS AND PROPERTIES.

Why? : they allow you to access methods and properties of a class without an instance of the class. For example : Math.PI / Math.pow() – no need instance to access these values or functionality.

In JS : static defines a static **method** in javascript

In TS : we can add static properties as well.

These are mostly utility functions, such as the ones used to create objects or clone objects.

The static properties and methods have to be accessed via “classname.staticmethod” from the  
 non-static methods, if needed.

ABSTRACT CLASS.

* An other way to adding methods to classes, which you plan to inherit from.
* In the case of inheritance: the derived class will override the methods, to provide their own implementation.
* Why ?   
  When you cannot provide a general method, but this method exists : But the inherited class will need to provide their own implementation [because we cannot provide default implementation in the base class].

**abstract** class **Institute**{

// you have to initialize it here or else the constructor must

private insID:string = '0';

constructor(protected insName:string){

}

set instituteID(ins:string){

this.insID = ins;

}

get instituteID(){

return this.insID

}

//the class must be 'abstract' if the function is + its return type must be explicitly mentioned.

//**abstract** describe():()=>void ;

**abstract** describe():void;

}

class **CoachingInstitute** **extends** Institute{

constructor(private subjects:string[]){

super("AppleCoaching Institute");

}

**describe**(){

console.log("This is a Coaching Institute class")

}

}

* Abstract can be useful if you want to enforce that all classes based on some other base class share some common method/property + we don’t want to provide the concrete functionality/implementation in the base class : The inheriting class has to do that.
* Abstract classes cannot be instantiated, They are only there to be inherited.

SINGLETON AND PRIVATE CONSTRUCTORS.

* Use of private constructor : there is a singleton pattern in object oriented programming.

This ensures you have only one instance of a particular class.

**abstract** **class** **website**{

// you have to initialize it here or else the constructor must

private websiteID:string = '0';

constructor(protected websitePostfix:string){

}

set websiteIDInfo(ins:string){

this.websiteID = ins;

}

get websiteIDInfo(){

return this.websiteID

}

//the class must be 'abstract' if the function is + its return type must be explicitly mentioned.

//abstract describe():()=>void ;

abstract describe():void;

}

class AppleWebsite **extends** website{

private static appleInstance : AppleWebsite;

// public matter :any = null; - this is possible

// static appleInstance : AppleWebsite = null ; - this is not possible

//only accessible inside the class : this is only possible with a static class

private constructor(private websiteName : string, private products:string[]){

super(".com");

}

describe(){

console.log(`this is a www.${this.websiteName}.${this.websitePostfix} website`)

}

static getAppleWebsiteInstance(){

//only return new instance if an old one does not exist

if (AppleWebsite.appleInstance || AppleWebsite.appleInstance !== undefined){

console.log('returning an existing apple instance')

return this.appleInstance;

}

else

{

console.log('creating apple instance')

this.appleInstance = new AppleWebsite("apple",['phones','ipods'])

return this.appleInstance;

}

}

//lets say there is only one apple website

}

const apple:AppleWebsite = AppleWebsite.getAppleWebsiteInstance();

const apple2:AppleWebsite = AppleWebsite.getAppleWebsiteInstance();

console.log(apple,apple2) // will be equal

apple.describe();

INTERFACES.

* Interface keyword exists only in typescript
* Interfaces are used to define objects.

interface Person{

name: string;

age:number;

greet(a1:string):void;

}

let user1 : Person;

user1={

name: "mike",

age:22,

greet : (a3:string)=>{

console.log('matter' + a3)

}

}

user1.greet(' mike');

USING INTERFACES WITH CLASSES.

* Sometimes interfaces and custom ‘type’ are used interchangeably
* When we define an interface we want to define the structure of an object.
* This is something like a contract a class has to implement and adhere to
* Interfaces are used to share functionality amongst classes – not regarding any concrete implementation.- only regarding the features a class should have.

interface Greetable{

name: string;

greet(a1:string):void;

}

// you can inherit only from one class

// but you can implement any number of interfaces

class Person implements Greetable{

name :string;

constructor(n1:string){

this.name=n1;

}

greet(abc:string){

console.log(`i am greetable ${abc}`);

}

}

//const user1: Person = new Person('Mike');

const user1:Greetable = new Person('Mike');

user1.greet("good morning");

Read only interface properties.

* A read only property in an interface means that : This property in whatever object you build : based on this interface, must only be set once. It cannot be changed after the object has been initialized.

// public or private access modifiers can not be added

// readonly can be added

**interface** Greetable{

readonly name: string;

greet(a1:string):void;

}

// you can inherit only from one class

// but you can implement any number of interfaces

class Person **implements** Greetable{

//readonly name :string;

name :string;

constructor(n1:string){

this.name=n1;

}

greet(abc:string){

console.log(`i am greetable ${abc}`);

}

}

// const user1:Greetable = new Person('Mike');

// user1.greet("good morning");

const user1: Person = new Person('Mike');

user1.name = "a";

EXTENDING INTERFACES.

interface Named {

readonly name :string;

}

interface Greetable{

greet(a1:string):void;

}

interface Greetable2 extends Named,Greetable{

greet(a1:string):void;

}

// you can inherit only from one class

// but you can implement any number of interfaces

// and you can extend from any number of interfaces

class Person implements Greetable2 {//Greetable,Named{

constructor(n1:string){

this.name=n1;

}

name :string;

greet(abc:string){

console.log(`i am greetable ${abc}`);

}

}

// const user1:Greetable = new Person('Mike');

// user1.greet("good morning");

const user1: Person = new Person('Mike');

user1.name = "a";

INTERFACES AS FUNCTION TYPES.

* Interfaces can also be used to define the structure of a function.

// type Addfn = (n1:number,n2:number) => number;

// let Add : Addfn;

// Add = (n1:number,n2:number) =>{

// return n1+n2;

// }

//typescript gets to know that : we want to use this interface as a function type

**interface Addfn {**

**(a:number,b:number):number;**

**}**

**let Add : Addfn;**

**Add = (n1:number,n2:number) =>{**

**return n1+n2;**

**}**

OPTIONAL PARAMETERS AND PROPERTIES.

* Optional properties can be defined in both interfaces and classes.

interface Named {

readonly name :string;

readonly age ? : number //optional properties in interfaces

outputName ? : string; // this property is optional, i.e the implementation class can skip to have this

}

interface Greetable{

greet(a1:string):void;

}

class Person implements Greetable,Named {

constructor(nam:string,ag ?:number){

this.name=nam;

if(ag){

this.age = ag;

}

else

this.age = 0;

}

name :string;

age ? : number;

greet(abc:string){

console.log(`i am ${this.name} and of ${this.age} years old`);

}

}

const user1: Person = new Person('Mike',22);

user1.greet("fff")

--------------- ADVANCED TYPES ---------------------

INTERSECTION TYPES.

**Why ?** They allow us to combine other types.. for example object types or built in javascript data types.

type Admin = {

name:string;

roles : string[];

}

type Employee = {

name:string;

startDate : Date;

}

type ElevatedEmployee = Admin & Employee;

//This object will contain both the object properties.

const ee: ElevatedEmployee = {

name:'max',

roles : ['admin','anotheradmin'],

startDate : new Date()

}

//THIS INTERSECTION TYPES ARE CLOSELY RELATED TO INTERFACE INHERITANCE ?? why?

interface Admin2 {

name:string;

roles : string[];

}

interface Employee2 {

name:string;

startDate : Date;

}

**// we can use an intersection type on interfaces as well**

//type ElevatedEmployee2 = Admin2 & Employee2;

interface ElevatedEmployee2 extends Admin2 , Employee2{

}

//The above code and the below code have the same effect

const ee2: ElevatedEmployee2 = {

name:'max',

roles : ['admin','anotheradmin'],

startDate : new Date()

}

//along with object types we can use with inbuilt data types

type combinable = string | number;

type numeric = number | boolean;

type universal = combinable & numeric;

//NOTE :UINIVERSAL IS OF TYPE NUMBER SINCE THAT IN THE ONLY COMMON ONE.

//IN CASE OF UNION TYPES : THE TYPES THAT THEY HAVE IN COMMON

// IN CASE OF OBJECT TYPES : COMBINATION OF ALL OBJECT PROPERTIES.

TYPE GUARDS.

* Type guards help us in union types. It is good to have the flexibility, often we need to know which exact type we are getting at run time.
* What is it? It simply an approach of checking which property or method exists before you use it.
* For objects : instanceof / in
* For other : typeof

// **Example : type guard using typeof operators**

type combinable = string | number;

type numeric = number | boolean;

type universal = combinable & numeric;

//NOTE :UINIVERSAL IS OF TYPE NUMBER SINCE THAT IN THE ONLY COMMON ONE.

//IN CASE OF UNION TYPES : THE TYPES THAT THEY HAVE IN COMMON

// IN CASE OF OBJECT TYPES : COMBINATION OF ALL OBJECT PROPERTIES.

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

//with out the typecheck - typescript does not know your intention - so it shows an error

if(**typeof** a === 'string' || **typeof** b === "string"){

return a.toString() + b.toString()

}

return a+b;

// tS knows if we have number we end up last : all of this is called typeguard.

}

// **ANOTHER EXAMPLES**  
type Admin = {

name:string;

roles : string[];

}

type Employee = {

name:string;

startDate : Date;

}

type ElevatedEmployee = Admin & Employee;

//OR

//interface ElevatedEmployee extends Admin , Employee{

//}

type unknownEmployee = Admin | Employee;

function PrinTEmployeeInfo(ee : unknownEmployee){

console.log(`Name in object : ${ee.name}`); // this is allowed by typescript

//console.log(ee.roles) // Ts will not suggest.. as it is not sure which kind of object param we received above

//console.log(ee.startDate) //TS will not suggest.

// if we use the typeof typeguard - no use

//Why ? the 'typeof ee' will yeild 'object' at runtime -

// -- and it does not tell us if we have this property or not

// if (typeof ee === 'object'){.....

// }

// further more we can check if the type is employee or not - typeof ee === 'employee'

// - because java script does not know anything about employee.

// - as the above code is executed at run time and javascript

// so a different if check can be used

if (typeof ee === 'object'){

if ('roles' in ee){

console.log('we got a admin object' + ee.roles) // this will work

}

if('startDate' in ee) {

console.log('we got a employee object' + ee.startDate) // this will work

}

}

// But the disadvantage of the above method is that - no mistakes must be made in the properties

// + for every object type we must have a if check

}

PrinTEmployeeInfo({

name:'mike',

roles : ['cricket coach','swim coach'],

})

PrinTEmployeeInfo({

name:'bike',

startDate : new Date()

})

// when working with classes we can use the instanceof typeguard.

class Car {

drive(){

console.log('driving in car');

}

}

class Truck {

drive(){

console.log('driving in Truck');

}

loadCargo(){

console.log('loading cargo in a truck')

}

}

type vehicle = Car | Truck;

const v1 = new Car();

const v2 = new Truck();

function useVehicle(veh:vehicle){

veh.drive();// we can call this as this always exists

// veh.loadCargo()

// this can not be called because only a truck has that and a car does not have it ...

// -some kind of typeguard needed again as shown below

**if('loadCargo' in veh){**

**veh.loadCargo();**

**}**

//Another way of doing it - instanceof is a normal operator built into vanilla javascript.

// note that javascript does not the truck type - but classes in the end are translated to constructor functions

// then TS will be able to find out if veh was created based on the Truck constructor function

// if some kind of check was to be made based on interface names then we can not make such check

// because interfaces will not be compiled to any javascript code - so cant use them at runtime

// we can do that for classes because JS supports classes and constrcutor functions.

**if(veh instanceof Truck){**

**veh.loadCargo();**

**}**

}

useVehicle(v1);

useVehicle(v2)

DICSRIMINATED UNIONS.

* Why ? : something that helps you with typeguards.
* What ? : it is a pattern which you can use when working with union types – that makes implementing typeguards easier. It is available when you work with object types. In this we give an extra property.

interface Bird {

type : 'bird'; // type must hold a string which must be bird.

flyingSpeed : number;

}

interface Horse{

type : 'horse';

runningSpeed : number;

}

type Animal = Bird | Horse;

function moveAnimal(an:Animal){

//method 1 - more check needed for more interfaces as you add in to your type + property names could be misspelled

// if('flyingSpeed' in an){

// console.log(`moving with speed ${an.flyingSpeed}`)

// }

// else

// console.log(`moving with speed ${an.runningSpeed}`)

//can not use instance of as we aer working with interfaces

switch(an.type){

case 'bird':

{

console.log('its a bird')

break;

}

case "horse" :

{

console.log('its a horse')

break;

}

}

}

moveAnimal({type:'bird',flyingSpeed:10})

moveAnimal({type:'horse',runningSpeed:10})

TYPECASTING.

WHY ? it helps you tell typescript that some value is of a specific type – where TS is not able to detect it on its own, but we know that it’s the case.

//const para = document.querySelector('p')

//TS find out that as you selected by element - it is a paragraphhtmlelement or null

const para = document.getElementById('para-id')

//TS does not dive into the html code - it recognises it just as a HTMLELEMENT

// for a paragraph we might do stuff we we would do with any other HTL element - for a para it might not matter much

//but if its a input

//const inpu = document.getElementById('input-id');

// inpu.value = "hi there"; // type script does not allow this

// typescript does not allow this..

// err 1 : object is possibily 'null'

// err 2 : property 'value' does not exist on type 'HTMLElement'

// we we have to tell TS that 'inpu' is not null but also an 'html input element'

//WHY ? THIS IS WHAT WE CAN DO WITH TYPECASTING.

// THIS IS POSSIBLE because we included the dom lib in the tsconfig libs

//const inpu2 = <HTMLInputElement> document.getElementById('input-id')!;

//inpu2.value="matter2";

// for react projects

const inpu3 = document.getElementById('input-id')! as HTMLInputElement;

inpu3.value = "matter3"

//NOTE : we are forcing TS to use this type and we are responsible for the consequences

! : this tells Ts that the expression in front of it will never yield null.

Alternatively if you are not sure about it, do an if check below.

INDEX PROPERTIES.

WHY ?   
A feature that helps to write flexible code.  
A feature that allows us to create objects that are more flexible regarding the properties that they hold.   
If we need an object where we are pretty clear regarding the value type and we don’t know in advance how many properties we will have and name of the properties . – such a scenario we use index types.

interface ErrorContainer {

//id:string;

//id2:number; // this can not done all should be string : string

[prop: string] : string;

}

// we dont know the property name

// we dont know the property count

// we just know that every property that is added 'to this object which is based on error container' -

// must contain a property name which can be interpreted as a string and the value for that proerty must be a string

const err:ErrorContainer = {

email:'not a valid email',

username: 'username already taken'

}

// If we dont know in advance which exact peroperty name we will have in there

// for example for a form in page 1 we might know - but what if we want to use this container for all forms in the website..!

// another thing is if username is valid and only the email is not - do you want an object with all of these properties ...?

// finally with this we will have an object with properties for which we have an error so that we can loop through them using for in..

FUNCTION OVERLOADS.

Why ?

What ? multiple possible ways of calling a function, by providing different number of parameters.

type Combinable = number | string;

//this below line means if we call this function - and both arguments are numbers - thenthis function returns a number

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

function add(a:string,b:string):string

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

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

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

if(typeof a === "string" || typeof b==="string"){

return a.toString() + b.toString();

}

return a+b;

}

//const val = add('abc','def') ;

// we can not call string functions on the result.

// we know that the result will hold a string - but Ts know it might be 'string or a number'

// we can do type casting

// add('abc','def') as string;

const val = add('abc','def');

val.split(''); // the split auto suggestion is listed here.

OPTIONAL CHAINING.

// sometimes we are not sure if the above object comes from outside.

// so in java script we do

// if(emp.address && emp.address.line1) ...emp.

// In TS we can do using optional chaining

const val2 = emp**?.**address?.line1;

NULLISH COALESCING.

Why ? : helps you deal with nullish data.

What ? : if you have some data and if you are not sure whether it is null ..undefined .

// sometimes we are not sure if the above object comes from outside.

// so in java script we do

// if(emp.address && emp.address.line1) ...emp.

// In TS we can do using optional chaining

const val2 = emp?.address?.line1;

let userInput = null ;// TS knows this is null, but if we are getting this from a backend.. Ts will not know at all.

//now if store this data which we recieved from other source into a local variable

const localvariable = userInput || 'DEFAULT';

// the problem with this above statement is if userInput is empty string '' then ...

// userInput will be treated as falsy and still DEFAULT will be assigned

console.log(`val is ${localvariable}`) // prints default when 'userInput = null' and 'userInput = '' '

//but if you want to keep the userinput unless it is null or undefined - we fail so far

userInput = null ;

const localvariable2 = userInput ?? 'DEFAULT';

console.log(`val2 is ${localvariable2}`)

userInput = "";

const localvariable3 = userInput ?? 'DEFAULT';

console.log(`val2 is ${localvariable3}`) // empty string is valid

GENRIC TYPES ------------ONLY IN TS---------

BUILT IN GENERICS.  
  
//array is also generic

const names:Array<string> = ['mike','bike']

// array type also stores another type of data in them

//so - below line is possible

names[0].split // can all string methods.

// promises - another generic type

// by default if you hover you get : const prom:Promise<unknown>

// this is right - const prom:Promise<string>

//promise just like an array - works together with other types -

// because eventually it return some data of some type..

const prom:Promise<string> = new Promise((resolve,reject)=>{

setTimeout(() => {

resolve("returning a string - hi there")

}, 2000);

})

prom.then(data=>{

data.split // string methods can be called.

})

GENERIC FUNCTIONS.

function merge(a:object,b:object)

{

return Object.assign(a,b)

}

console.log( merge({name:"mike"},{age:20}))

//the above statement works - but

const mergedobj = merge({name:"mike"},{age:20}) ;

//mergedobj.name // .. but this will not work

//alternatively it can be done through typecasting --but cumborsome + you are hardcoding the output..

const mergedobj2 = merge({name:"mike"},{age:20}) as {name:string,age:number};

console.log(mergedobj2.name); // this will work

//LETS USE A GENERIC FUNCTION HERE

function mergeAgain<**T**,**U**>(a:T,b:U) //hover - T & U

{

return Object.assign(a,b)

}

const mergedobj3 = mergeAgain(**{name:"bike"},{height:30}**);

console.log(mergedobj3.height); // this will work – auto suggest

WORKING WITH CONSTRAINTS.

Use ? when we pass parameters to generic functions – we don’t want them to be anything of any type-we might want to be anything but of ‘object type’. [using the extends keyword]

Constraints helps you work with generics in a better and optimal way.

//we dont care about the extact type -here

function mergeAgain<T extends object,U extends object>(a:T,b:U) //hover - T & U

{

return Object.assign(a,b)

}

//we dont care about the extact structure of the object

// - we must stress that they must be objects of any kind - not anything of any type

//as someone could pass something which is not object like just 30 instead of {age:30}

const mergedobj3 = mergeAgain({name:"bike"},30); // TS complains

const mergedobj4 = mergeAgain({name:"bike"},{age:30}); //TS does not complain as its an object

console.log(mergedobj4); // this will work

ANOTHER GENERIC FUNCTION.

interface lengthy {

length :number;

}

//Initially typescript wont recognise length on that generic type

//but after extending from lengthy - the err goes +

// we explicitly say that what ever we get has a length property

function countAndDescribe<T extends lengthy>(element: T) : [T,string]

// we set the return type to be a tuple - we also say the typoe of the first ele

{

let desc:string = "";

if(element.length > 0) {

desc= `Got ${element.length} elements`

}

else

{

desc = "Got no value"

}

return [element,desc]

}

console.log(countAndDescribe("matter"))

console.log(countAndDescribe(["mike","bike"]))

console.log(countAndDescribe(20))

//the string will be converted to an object and string properties will be added but not a number

KEYOF CONSTRAINT (Another type of constraint).

// object is the first param

// we pass a key as the second param and the object must ...

// ..contain that second param as a key- as we want to extract the value of that key

function extractAndConvert<T extends object,key extends keyof(T)>(obj:T,prop:key){

return obj[prop];

}

console.log(extractAndConvert({"name":"mike",age:30},"name") )

GENERIC CLASSES.

// one or more types can be used here

class Storage2<T1 extends string | number> {

private data : T1[] = [];

// GENERIC TYPES CAN BE USED OVER METHODS ALSO

addItems(item:T1){

this.data.push(item)

}

removeItems(item:T1){

// this does not work well when working with non-primitive value like objects

this.data.splice(this.data.indexOf(item),1)

}

getItems():T1[]{

return [...this.data]

}

}

const textStorage = new Storage2<string>();

textStorage.addItems("mike")

textStorage.addItems("bike")

textStorage.addItems("hike")

textStorage.removeItems("bike")

textStorage.removeItems("mike")

console.log(textStorage.getItems())

// const objStorage = new Storage2<object>();

// objStorage.addItems({name:"one"})

// objStorage.addItems({name:"two"})

// objStorage.addItems({name:"three"})

// objStorage.removeItems({name:"two"})

// //console.log(objStorage.getItems())// one and two will still be there..NOT WORKING

// // as index of works with references of objects - below will work

// const objStorage2 = new Storage2<object>();

// const one = {name:"one"}

// const two = {name:"two"}

// const three = {name:"three"}

// objStorage2.addItems(one);objStorage2.addItems(two);objStorage2.addItems(three)

// objStorage2.removeItems(two);console.log(objStorage2.getItems())

// so for Storage2 to work - it should only be applicable to primitive types

GENERIC UTILITY TYPES – built in types that utilize generic types.

Why ? extra type safety – extra flexibility

//PARTIAL UTILITY TYPE

interface CourseGoal{

title:string;

description:string;

completeUntill:Date;

}

function createCourseGoal(title:string,description:string,date:Date) : CourseGoal{

//All in one step

// return {

// title,

// description,

// completeUntill:date

// }

//another way - but {} is not of type CourseGoal : so partial type comes in handy here

//let courseGoal:CourseGoal = {};

// all the properties become optional now

let courseGoal: Partial<CourseGoal>= {};

courseGoal.title = title;

courseGoal.description = description;

courseGoal.completeUntill = date;

return courseGoal as CourseGoal;

}

//READ-ONLY UTILITY TYPE

//const names:string[] = ["one","two"]

//wanna lock the above array

const names:Readonly< string[] > = ["one","two"]

//push and pop will not allowed after the readonly generic utility is used

// can be used on objects also

names.push("three"); //wont work

--------------- DECORATORS .

* WHY ? : you can write code which is easier to use by other developers
* What : it’s a function that you apply to something in a certain way

//FIRST WAY TO CREATE A DECORATOR

function Logger(cf:Function){

//console.log("Logging")

//console.log(cf)

}

@Logger

class Person1 {

name:string = "Max"

constructor(){

//console.log("creating person object")

}

}

const \_per = new Person1();

//console.log(\_per)

//--------------------

// DECORATOR FACTORY - returns a decorator function +

// but allows us to configure it - when you assign it as a decorator to something.

// we have a function that returns a new function

function Logger2(logString:string){

return (cf:Function)=>{

console.log(logString)

console.log(cf)

}

}

//This way we can customize the values the decorator function uses with our factory function.

// we are executing a function that will return a decorator function

@Logger2('LOGGING-PERSON')

class Person2 {

name:string = "Max"

constructor(){

console.log("creating person object")

}

}

const \_per2 = new Person2();

console.log(\_per2)

MORE DECORATORS

function WithTemplate(htmlCode:string,domId:string){

return function(cf:any){

//wanna render html code in a domid

const root = document.getElementById(domId);

if(root) {

console.log("matter")

root.innerHTML = htmlCode;

}

const obj = new cf();

const para = document.getElementById('para');

if(para)

{

para.textContent = obj.name;

console.log(obj.name)

}

}

}

@WithTemplate('<h2>Hi There </h2>','app')

class Person2 {

name:string = "Max"

constructor(){

console.log("creating person object")

}

}

const \_per2 = new Person2();

console.log(\_per2)

ADDING MULTIPLE DECORATORS.

function Logger2(logString:string){

console.log("Logger - decorator Factory")

return (cf:Function)=>{

console.log(logString + " Logger2 - decorator function")

console.log(cf)

}

}

function WithTemplate(htmlCode:string,domId:string){

console.log("WithTemplate - decorator Factory")

return function(cf:any){

console.log('WithTemplate - decorator function')

const root = document.getElementById(domId);

if(root) {

root.innerHTML = htmlCode;

}

}

}

@Logger2('LOGGER DECORATOR') // function execution 1st // decorator execution second

@WithTemplate('<h2>Hi There </h2>','app') //function execution second // decorator execution first

class Person2 {

name:string = "Max"

constructor(){

console.log("creating person object")

}

}

PROPERTY DECORATORS.

// This decorator is executed when the class definition is registered by javascript.

// Which arguments the decorator gets depends on where you add it.

function LogPropertyDecorator(target:any,propname:string){

// Arguments info

// if we add a decorator to a proeprty the decorator receives two arguments.

// first arg : target of the property

// for an instance property - the target will be the prototype of the object that was created

// for a static property - the target will refer to the constructor function instead.

console.log('This is a property decorator')

console.log('target is : '+ target)

console.log(target)

console.log('Property name is : ' + propname)

}

class Product {

@LogPropertyDecorator

title:string;

price:number;

set sPrice(pri:number){

if(pri>0)

{

this.price = pri;

}

else{

throw new Error("Invalid price - should be positive")

}

}

get gPrice(){

return this.price;

}

constructor(titl:string,pric:number){

this.title= titl;

this.price=pric;

}

getPriceWithTax(tax:number){

return this.price \* (1+tax)

}

}

ACCESSOR AND PARAMETER DECORATORS AND OTHER DECORATORS.

function LogPropertyDecorator(target:any,propname:string){

console.log('This is a property decorator')

console.log('target is : '+ target)

console.log(target)

console.log('Property name is : ' + propname)

}

function LogAccessorDecorator(target:any,name:string,propdetr:PropertyDescriptor){

//second argument : name of the member

console.log('accessor decorator');

console.log(target);

console.log(name); //name of the accessor itself

console.log(propdetr); // property descriptor

//writable/enumarable/configurable..etc

}

function LogMethodDecorator(target:any,name:string,propdetr:PropertyDescriptor){

console.log('METHOD decorator');

console.log(target);

console.log(name); //name of the accessor itself

console.log(propdetr); // property descriptor

}

function ParameterDecorator(target:any,name:string,parampos:number){

console.log('PARAMETER DECORATOR');

console.log(target);

console.log(name); //name of the method in which the parameter is used

console.log(parampos);

}

class Product {

@LogPropertyDecorator

title:string;

price:number;

@LogAccessorDecorator

set sPrice(pri:number){

if(pri>0)

{

this.price = pri;

}

else{

throw new Error("Invalid price - should be positive")

}

}

get gPrice(){

return this.price;

}

constructor(titl:string,pric:number){

this.title= titl;

this.price=pric;

}

@LogMethodDecorator

getPriceWithTax(@ParameterDecorator tax:number){

return this.price \* (1+tax)

}

}

RETURNING AND CHANGING A CLASS IN A CLASS DECORATOR.

A class decorator or a method decorator is capable of returning something.

i.e a decorator function returning something.

What you can return from a decorator depends on which kind of decorator you are working with.

If it is a *class decorator* we can return a new constructor function.

With the below decorator code – we run code when a class is instantiated but not when the class is defined.

function Logger2(logString:string){

console.log("Logger - decorator Factory")

return (cf:Function)=>{

console.log(logString + " Logger2 - decorator function")

console.log(cf)

}

}

**function WithTemplate(htmlCode:string,domId:string)**

**{**

console.log("WithTemplate - decorator Factory")

// we are trying to replace the class/constructor function - to which we added our decorator ...

// .. with a new constructor function

// .. now the console log will be printed - only if you instantiate the object in which this decorator is used.

**return function<T extends { new(...args:any[]) : {name:string} } > (originalConstructor:T)**

// we have to make sure that what ever T extends from is a CONSTRUCTOR function by using an object type

// new - an object which can be newed

// the new method which T is based on will get any no of arguments.

// and the new function will return an object which contains a name property

// learn about INTERFACES AS FUNCTION TYPES if "new(...args:any[]) : {name:string}" confuses you

**{**

// We are returning a new class OR a constructor function.

// .. Which is based on the orignal constructor function [so that we preserve all the properties of the original class]

**return class extends originalConstructor**

**{**

**constructor(...\_:any[])**

**{**

**super();**

// Now the below code will only be executed - upon instantiation only

**console.log('WithTemplate - decorator function ' + this.name)**

**}**

**}**

**}**

**}**

//@Logger2('LOGGER DECORATOR') // function execution 1st // decorator execution second

**@WithTemplate('<h2>Hi There </h2>','app') //function execution second // decorator execution first**

**class Person2 {**

**name:string = "Max"**

**constructor(){**

**console.log("creating person object")**

**}**

**}**

**const \_per2 = new Person2();**

//console.log(\_per2)

OTHER DECORATOR RETURN TYPES.

We can return something on the decorators we add to accessors and methods.

* We can return a property descriptor in this case.

The decorators on properties and parameters also can return something – but TS will ignore it.

Returning on a method decorator.