TypeScript - 2022 Edition

~~Install typescript~~

① npm install -g typescript

tsc hello.t

(compile)

Overview :- Typescript adds

→ like Babel

Bhupendra Singh

TypeScript

JavaScript Superset, which is next level

** Browser, Node.js can't be access typescript.

→ typescript is a programming language or also a tool, it's a power Compiler which you run over your code to compile your TS code to JS.

⇒ Install typescript

① npm install -g typescript

Compile

tsc hello.ts

Overview :- Typescript adds

→ Types!

→ Next-gen javascript Features (compiled down for older browser) → like Babel

→ Non-javascript Features like interfaces or generics

→ Meta-programming Features like Decorators

→ Rich Configuration options

→ modern Tooling that helps even in non-Typescript projects.

Typescript Course

- Typescript Basics
- Compiler & Configuration Deep Drive
- Working with Next-gen JS Code
- Classes & Interfaces
- Advanced Types & Typescript Features
- Generics
- Decorators
- Time to practice - Full project
- Working with Namespaces & modules
- Webpack & Typescript
- Third-party Libraries & Typescript

→ Javascript use run time type while Typescript use static type.

Visual IDE Extensions :-

1. ESlint
2. Material Icon Theme
3. Path Intellisense
4. prettier
5. TSLint

Working with Types :-

Core type :-

1. number → 5, 0.5, -5
2. ~~String~~ String → 'H', "H", `Hi`
3. Boolean → true, false
4. Object.....

(*) In typescript, If a value type is string, then ~~we~~ ~~cannot~~ it should be same.
but in javascript, If a value type is string, then it can be changed as per logic.

Ex Let value1 : number = 1 → this is not good for practice

good way: Let value1 : number ;

function add(n1: number, v1: string){
return n1 + v1
}

(4) Object → {age : 30}

E.g

```
Const person : object = {
    name : 'Bhupi',
    age : 24
}
```

showing an object type

Or

```
Const person : { } = { —"— }
```

or

```
Const person : {
    name : string,
    age : number
} = {
    name : 'Bhupi',
    age : 24
}
```

properties are specified

Define all properties types

otherwise It will give an error.

Type and Type Assignment are not part of JavaScript

5 Arrays

let a = string[] → array of things

E.g a = ['abc', 'def']

a = ~~'string'~~ ; will give an error

a = ~~['abc', 2]~~

used a : any[]

6 Tuple

only in TS

→ store diff types of values in single variable

→ [', 2] → Added by typescript : Fixed length array.

(Also fixed type) once initialized

E.g let role = [2, 'new']

Property = (string | number)[]

tuple type = [number, string]

Push is an exception, which tuple allows.

once we defined a certain position then we can't change the type position

let tup : [string, number, Boolean]

⑦ ENUM → enum {NEW, OLD} → Added by TS ⑥

only in TS

Automatically enumerated global constant identifiers

Basically human readable numbers or constants

Enum is custom type, always start with upper case letter.

Labels

E.g enum Role {

ADMIN, READ_ONLY

};

You can provide Assign ADMIN=2

use

Const value = Role.ADMIN

Default start from zero (0)

⑧ Any → * → Any kind of value, no specific type assignment.

Disadvantage, because it takes all advantages that javascript gives you.

* used → when we donot know that which type of data are stored

9 UNION

E.g. function add(first: number | string){
return first
}

union

↳ we can use more than 2 as well

we can also provide our specific type

a | b | c | d

let a = 'abc' | 'def'

10 Literal

used to check the provided type with some specific condition or based on provide values.

→ Specically useful when used in conjunction with Union types.

let a: 'car' = 'abc';
a = 'bike' → error

11 type Alias

→ we can create our custom type with Union

let a = number | string
let b = Boolean | number

type New = number | string | Boolean

let a = New

Function Return Types and Void

→ Void return type mean that function does not have any return type.

any

In Javascript, If we use the return value of the function that does not return anything, we get Undefined as a value.

→ In typescript variable can allowed ~~undefi~~ undefined as a type,

But function does not allows undefined as a type.

use Void for all scenario's

Function type

```
function add(A, B){ return A+B; }

let value : Function;

value(1,2);
```

Return type for fn

→ Function types are types that describe a function regarding the parameters and the return value of the function.

→ A function type is created with '⇒' (annonetion)

with no parameter → return number

E.g

```
let val : () ⇒ number
let val2 : (a: number, b: number) ⇒ number;
```

Function Types and Callbacks

→

```
function add(n1: number, n2: number, cb(num: number)
                                    ⇒ void){
    const result = n1 + n2;
    cb(result);
}
```

⇒ `add(10, 20, (result) => {});`

The unknown type

let val : unknown;

val = 5;

val = 'abc';

Can't do any operation

val === 3

~~val * 3~~

~~val + 3~~

let user : string;

user : val → error

~~val = user~~

★★ But any will work for all.

⇒ Anything is assignable to unknown but unknown is not assignable to anything

The never type

: never

```
function generateError(a: number, code: string) {
    throw {message: a, errorCode: code};
}
```

→ generateError('An error occurred!', 500);

This function does not have any return type.

→ Can't assign anything to never type.

Typescript Compiler

Before :- `tsc a.ts`
↳ for run a.ts file

For Automatic Compiler → tsc a.ts --watch
or
tsc a.ts -w
} for single file

→ tsc --init → for all the files
generate tsconfig.json file
After this, Run
tsc
tsc --watch

→ In tsconfig.json file, we can use exclude and include for ~~run~~ ~~used~~ compile or not compile ~~unused~~ the file during ~~run time~~ compile time.

In the end

"exclude" : ["a.ts"]

for all file and's with

"*.dev.ts"

"**/*.dev.ts"

parent folder

→ include : ["a.ts"] → only compile this file.

→ files : [–] → for particular file

Debug

Extension → ESLint,
Debugger for chrome

✓ Sourcemap : true

launch.json file

Modules | let, const & var

① function add(a1: number, a2: number) {
var a = 4;
console.log(a);
}
console.log(a);

→ if (true) {
var a = 2
}
log(2) ⟶ error in b

Because with var javascript does not know any scope other than function in globally.

Arrow function

Const add = (a: number, b: number) => a+b;

⇒ [only for single line return statement]

for 1 param

Const add : (a: number | string) => void = otp => otp;

→ we can also provide default value here

string = 'abc';

spread operator

Const a = ['a', 'b'];

Const b = ['c'];

→ Const b = ['c', ... a]

or

→ b.push(... a)

→ used to pulling out elements from arrays or objects.

Const a = {
name: 'abc'
}

Const b = {...a}

Rest operator

→ Expected all incoming value through rest operator And Default type will be any

```
const add = (...num) => {
        (...num: number[])
return num.reduce((curRes, curVal) => {
        return curRes + curVal;
    }, 0);
};
const numbers = add(1,2,3,4,0.6,9.2);
```

→ We can also combined with types. (If the list of element is not unlimited)

(... numbers: [number, number, number]

Arrays & Object Destructuring

```
const a = [1,2,3,4,5];
const A = a[0];
const B = a[1];
```

```
const [A, B] = a;
```

→ Same for objects

```
const a = {
    name: 'abc',
    age: 20
};
```

```
const {name, age} = a
```

And Also You can reassign the values like

```
const {name: first, age} = a
```

Classes & Interfaces (what is a class property!)

Ans: Basically a variable in a class

Classes :- work with (real-life) Entities in your code

→ Blueprints for objects

→ The object of an Instance of class, so that we can quickly replicate multiple objects with the same structure and same methods based on same class.

So classes help us to find how objects should look like which properties and methods they have.

Constructor is used to initialization objects that you building.

→ In ES5, If you initialize the Constructor, then you will be able to see Construction function while initialize. (Vanilla)

this → This typically refers back to the concrete instance of this class that was created and (.) dot notation we can access all the properties and method of this instance.

Ex

```
Class ABC {
    name : string
    constructor (n : string) {
        this.name = n
        }
    Describe () {
        Console.log("____", this.name)
        }
    }

Const new = new ABC("OK");

new.describe();   // ____ OK

but

Const ab = { Describe : new.describe } — (1)
ab.describe()   // undefined
```

** Because we are adding Describe property to that object which is created with object literal not based on that class but just as a dummy object.

⟹ $\boxed{\text{Describe(this : ABC)}}$

this will always prefer to the instance that based on the ABC class

Then const ab = { name: 'A', describe : new. Describe }

↓

from eq. ①

Q ⇒ If you want to private any method under the class, ~~tne~~ so that we can't use outside of the class.

Then use 'private' keyword for that.

Q ⇒ JS does not have publicand private concepts.

⇒ You can also assign your params as private or public like

⇒ Constructor (private id: string, public name: string)

↳ which arguments to constructor takes, which has such "access modifier"

Readonly (does not accept in JS)

You can only use once during initialization

Constructor (private readonly id: string)

↳ Can't assign to any other variable / arguments, because of read only.

Inheritance

[Private properties, can't access/inherit]

⇒ When you inherit the class through extend keyword the class which we inherit automatically gets everything the base class including it's constructor.

But if you want, then you can add your own constructor.

** [Can't inherit multiple classes]

You have to use "super" in your inheritance class.

```
super();
```

** Before you do anything with "this" keyword.

```
Constructor (id: string, public admins: string[]){
                                          New
}
```

override

"Protected" modifier

~~use when you~~ This will work as ~~pri~~ private modifier, but you can access this in inherit class also.

Getters and Setters

Private privateArgs : string

① get methodName() {
return this.privateArgs ;
}

Now we can access the values outside with the help of getters.

② set methodName (value: string) Args required
(value) {
———"———
}

* Need to set this method

✓ `a.methodName = 'ABC'`

** These methods are used to encapsulate logic

Static methods

→ this property will not work inside constructor

→ we call directly any method ~~from~~ on the class without a new keyword. we used class as grouping mechanism.

→ Also we can use/call any property. Concrete implementation parts.

If you want to use static properties inside the constructor then you ~~need~~ to ~~use~~ would have to use class name there to access it

Abstract

abstract abc(this: ClassName): void;

⇒ is useful if you wanna enforce that all class based on some other class share some common methods properties / abstract properties, ~~but abstract~~

⇒ But at the same time you wanna make sure don't have provide the concrete value the concrete implementation and the base class, but inheritance class has to do that

A class that can't be instantiated but has to be extended

⇒ Abstract class marked as "Abstract" with this keyword

Private Constructor / Singleton Pattern

→ only one instance of the certain class

if it is private then you can't use outside for that you need to ~~use~~ static method along with getters methods to access it

Interface [→ You can only use "read only"]

→ Describe the structure of an object.

→ You can use ~~of~~ first letter of interface upper or smaller both.

→ Interface can't have an initializer.

* Diff between Interface and type Keyword

→ * Interfaces can ~~be~~ only be to describe structure of an object, you can use type as ~~well~~ well.

but in custom type you can also store our thing like union type.

→ You can implements more than one interface by comma separate.

→ Class person implements A, B { }

** interface Vs abstract

~~As~~ interface has no implementation details where as abstract classes can be mixture of you have to override and concrete implementation parts.

Intersection type

It is like inheritance, when you can assign multiple interface to one

type — ①
type — ②

[Common will be merged into one]

✓ const A = ① & ②

Type Guards

This will check/verify the type/fields under the Interface

①
```
type A {
  name : string
}
```

②
```
type B {
  age : string
  date : date
}
```

```
type C = A | B ;
```

* It also work in class

```
function pop(abc : C) {
  * if ('date' in abc) { }      — ①
  * if ('name' in abc) { }
```

OR

```
  * if (date instanceof B)      — ②
```

* but in class

Discriminated unions

①
```
interface A {
    type : 'car',
    speed : number
}
```

②
```
interface B {
    type : 'xyz',
    newSpeed : number
}
```

③
```
type C = A | B

function ABC (val : C) {
    let speed :
    switch (val.type) {
        Case : 'car' → you can't give others
        Case : 'xyz'
```

type assignment

~~one~~ one common would be there that described the "type" of object.

type Casting

used to provide the exact type to the used values / arguments / syntax / parameters ...

```
Const a = document.getElementById('___');
(a as HTMLInputElement).value = '___';
```

Optional Chaining

```
if (a?.b?.c)
```

It just a optional check that will use to ~~get~~ check the value

Nullish Coalescing

```
const a = undefined
const b = a ?? 'DEFAULT'
```

Generics → They give us flexibility, combined with type safety.

```
const names: Array<string> = [];
```

If you are not sure what type of data will be there inside array.

Constructor function

```
const promise: = new Promise(resolve, reject) => {
                 }
```

Promise<string>

By default this type will be

Promise<unknown>

Create Generic function

```
function merge (a:obj, b:obj) {
  return Object.assign (a,b);
  }
const merged = merge({name: 'A'}, {age:30});
```

because this is specific object type and will return same

If you will try to assign ~~some~~ ~~no~~ same property again, then it will give an error.

merged.name = 'p' → Error

```
function merge <T, U> (objA: T, objB: U) {
return Object.assign (objA, objB)
}
```

→ convert into intersections

works → next page

out in class

~~Const merged = merge<string, number>~~

Const merged = merge<{name : string}, {age : number}>

(({name: 'A'}, {age : 20}))

→ (or not required)

Constraints

You can extend the specific types

function merge<T extends object>(obj A : T){}

The keyof Constraints

function A<T extends object, U extends keyof T>(
obj : T, key : U){
return 'value:' + obj[key]
}

A({name: 'A'}, 'name')

Generics in Classes

It will work for all primitive types like String, boolean and numbers,

but giveran error for object, because it store in new memory

Decorator

* tsc --target ES5 --experimentalDecorators

First class decorator

* tsconfig ~~Package~~.json
  ↳ experimentalDecorators : true
  ↳ without this we can't use decorator in our project.

→ Decorators runs when javascript finds your class definations, ~~on~~ ~~can~~ your constructor definition function not when you used that constructor for instiatite.

→ * Decorator : just a function that we can use with the help of "@". and this evaluate to a function that will be called at runtime with information about the decorated declaration.

E.g

```
function Logger (Constructor : Function) { }   — (1)
```

or

```
function Logger (log : String) {
    return function (Constructor : Function) { }   — (2)
}
```

→ Decorator factory is just a fn that return the expression

* @Logger — (1)
  @Logger('A') — (2)

* And it accept one argument "target"

→ * Decorator provide extra utility to developers, which others developers can use, for example to render something on the screen for a given class.

out in class

→ You can add multiple decorator to a class.

→ Bottom to up order But decorator factory from up to bottom.

or

① The expression for each decorator are evaluated top-to-bottom

② The results are then called as fn from bottom-to-top.

If you will return a class with extends from constructor then that will replace your original class and constructor function.

⇒ when you use add event listner and calling any class method then "this" will behave different. for that you need to bind the method call.

Autobind → "bind" method will take the current persons / objects.

→ You can also put a function for decorator

namespace

a.ts

```
namespace App {
  export ABC {
    ____
    ____
  }
}
```

b.ts

```
/// <reference path
  = "a.ts" />
```

then ↓,

```
namespace App {

}
```

just export the namespace from another file and then you can use in another file with reference import and the same interface name.

for namespace you need to change "module": "amd" in tsconfig.json file

and "outFile": "./dist/bundle.js"

→ **problem | dis Advantage :-**

when we manually added all the dependency to each file.

"es2015"

ES Modules

a.ts → .js

```
export ABC { -}
```

b.ts

```
import {ABC} from '../a.js'
```

after compiled → .js it ~~Because~~ convert into js

```
<script type="module" src="dist/app.js">
  </script>
```

① a.ts — Normal

export ABC {-}

② b.ts

import {ABC} from '..\a.js

or

import * as Val from '..\a.js'

~~Import~~ default Export

① a.ts

export default pack {-}

import CMD from 'a.js'

Because of default

Webpack

→ Webpack is a tool that will help us bundles our files together.

→ That will also ~~be~~ help us to reduce amount of http request by bundling code together.

→ It optimize our code

npm install webpack webpack-cli webpack-dev-server typescript ts-loader

- typescript: If ts version will change globally then automatically it will take
- ts-loader: ts to js
- Compile with the help of ts-loader

Configuration

tsConfig.json

① webpack and typescripter loader will use "target" : "ES6". to determine How your code should be transformed.

② rootDir commented, webpack will take over here

webpack.config.js

```
const path = require('path')

module.exports = {
    mode : 'development',
    entry : './src/app.ts',

    output : {
        filename : 'bundle.js',
        path : path.resolve-
            -(__dirname, 'dist'),
        publicPath : 'dist'
    },
    devtool : 'inline-source-map',
    module : {

        rules : [{
            test : /\.ts$/,
            use : 'ts-loader',
            exclude : /node_module/
        }]
    },
    resolve : {
```

when you use this then during import time no need to mentioned ".js" extension

Here you can also provide dynamic 'bundle.[contenthash].js' for create a unique hash for every build that will help you for caching and browser.

regular expression applied .ts

How to deals with ts files

→ next page

```
resolve: {
    extensions : ['.ts', '.js']
    }
};
```

"Source map" : true ★ This will also help us to debugging our code

★ package.json

```
"script" : {
    "build" : "webpack"
}
```

— 1

★ You can also use "webpack -dev- server" here

★ For production ★

```
const cleanplugin = require('clean-webpack-plugin')
```

```
mode : 'production'
publicPath —x (Not required)
devtool: 'none'
plugins : [
        new cleanplugin.cleanwebpackplugin()
]
```

delete all the in dist folder before new o/p is returning there

★ "build": "webpack --config webpack.config.prod.js"

— 2

→ Google geocoding API

① → types | google maps

* google map project