Typescript & ESNEXT [ES6]

setup typescript before start code.

Typescript : superset of javascript
- any valid javascript is typescript
- typescript can't be run directly on browser like js does
- can use ES6 & open source

~~# Variable & dela~~ # Variable & declarations

to run typescript you have first compile it

⇒ tsc index.ts -w → flag

this will return index.js file & now run it on node

⇒ node index.js → first convert ts → JS & run JS

Any valid (JS) in valid (TS)

⇒ tsc index.ts && node index.js

⇒ If you don't want compile each time

⇒ tsc index.ts -w ← good

⇒ this will detect change & recompile

Variable declaration (let if const)

→ previously var also used to create variables, but have problem with it

Say you use a loop

```
for(var i = 0; i<5; i++){
    code
}
```

so if you console

console.log(i) ⇒ (5) → value

but Let is valid for short duration and occupy memory

⇒ scope of let is small while var variables will stay in main scope of function

⇒ let keyword variable has small scope and not will able to show outside scope of let

⇒ we can access var outside it's declaration scope

Const

→ It is same like let, but const variable can't be reassigned.

→ you must have to give value while declaration

✓ `const a = 10;` `const a; a = 10` ⇒ not possible

⇒ you can reinitialise var variable ↱

Ex: `var i = 100` `var i = 1000` `var i = 10000`] no error

`let i = 0` `let i = 10` → multiple declaration give error same for const

⇒ let & const are block scoped variable

⇒ you can add values in case of const objects

⇒ internal states of const can be reassigned ↗

Primitive datatype

→ boolean → string → null → undefined

→ numbers

⇒ Same as javascript

⇒ to initialise a variable you have to use keyword

Ex: `let v : boolean;` created v which is of type boolean

similar to `int a;` in Java or CPP

⇒ to declare some primitive

```
let a: number;
let b: boolean;
let c: string;
let d : null ;
let f: undefined;
```

Such like this you can initialise a variable of specific datatype

just like javascript as we can change datatype of variable, you can't do it in typescript

⇒ you can't assign different datatype value to a specific variable

```
let a: boolean;
a = true;  ✓
a = 10;  X (Raise error)
```

```
let b: string
b = "hell"  ✓
b = true  X raise error
```

⇒ for number you can assign any type of num

```
let c: number
c = 10 ; c = -10 ; c = -10.8 ; c = 10.8888
c = 0b1010 , c = 0x123
```

⇒ you can create variable of multiple type

"|" ⇒ is used as or Imp

```
let g = number | boolean | string;
```

so g can now get value of number boolean string

"|" operator Imp

Strings "" ⇒ double quote '' ⇒ single quote

`` (back ticks)

```
a = `hello`
```

in typescript back ticks can be used to create string

Reference & special types

(A) void (B) ~~Array~~ (C) Array<type> (D) tuple

(E) objects (F) enum (G) any special

(H) ~~create a function of specific type~~ → now function

function name( ) : datatype { will have this datatype

}

~~for number~~

⇒ (A) Void ⇒ function with no return type value

Ex function hell() : void {

}

Void keyword used to create function with no return value

(#) Array

you have to declare it & then use it

let a: number[];

a = [10, 20, 30, 90]

⇒ if you declare a array of specific type you can use specific type value only in array

' if you not declare type previously then you can use

aa = ["str", 10, True]

↑ undefined so any type can live in same array

Another way

let a: Array<number>;

b: Array<string>;

you can declare array of multiple type

aa : Array<number | string | boolean>

or in common notation

cc : (number | boolean | string)[]

So :

(#) tuple

to create tuple

let t : [number, boolean, number]

so tuple t take 3 value & type should be in same order

⇒ a All value should be filled. Ex

let m : [string | number, boolean | null | string]

⇒ the tuple should have all value fullfilled else throw error

a = [number, boolean] → count of datatype in square bracket determines how many value should must be in tuple

⇒ not used so much

objects

```
let a = {
    name : "grs",
    mis : 111903033,
    func () {
        Console.log(this.name)
        console.log (this.mis)
        }
    }
```

⇒ object containing property & method

enum

to create specific set of values we use enum

```
enum daysofweek {
    Monday,
    tuesday,
    thursday
}
let a : daysofweek;
d = daysofweek.Monday
d = dayofweek.tuesday
```

such like that

⇒ Ex you can store math constants like (pi, e) in enum

any

let a : any

So a can be number, enum, object, boolean, array anything

⇒ If you want variable freely like we do in JS just initialise it with any and you can use any datatype for that variable

---

try to use any is most of case.

If you don't know specific type to use for a variable you can use any

---

function abc() : void {

}

⇒ a void function

function zab() {

}

⇒ can have (any) returntype

⇒ you have to use (: void) only if you want to create function which has no return value

---

Spread operator (denoted by ...) triple dots

Spread operator used to spread elements of array / object

⇒ merging multiple objects into one flat array/object

⇒ Ex

```
let a:number = 10;
```

→ initialise & declare at once

~~let b: string = "hgrs";~~

```
let c : Array<any> = [1,2,3]
let d : Array<any> = [4,5,6]
let f : Array<any> = [...c, ...d]
```

↓

```
f=[1,2,3,4,5,6]
```

(...) any valid for array type & object

So f array array we get value of c & d array

Spread in object

let c = {
name : "grs",
mis : 111903033
}

let d {

---

functions

function name(param1 : number, param2 ? : string) : number {

return 10;

}

raise error if value is not returned

param1 : number → parameter 1 should be a number

(? :) → signifies non-mandatory parameter

: mandatory parameter

Backticks (fstring in python)

used to create string → similar to Backticks

a = (` name : ${variable} `)

${} ⇒ used in typescript

let num = 199

b = ` the num is ${num} `

but ${variable} ⇒ It stringify thing like object array

Destructure ⇒ { }

→ Breaks structure of an array
→ used for importing statements

Ex: c = {
name : "grs"
mis : 111903033 }

{ name }
= c

So name will grabbed from (c)

=) destructure grabs some value from a structure like array, object

{name: n, mis: n} = c

alias change name

{name, mis} ← used for destruction

object

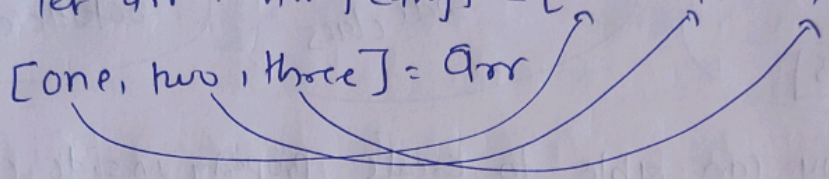
=) destructure used to grab values from object and array

destructing array

[ ] used

let arr : Array<any> = ["one", "two", "three"]

[one, two, three] = arr



Such like you can destruct array in typescript

Rest(...args) (like arbitary positional argument in Py)

=) Allow us to have indefinite argument in function in array

=) Can only used as last parameter.

=) You can fetch the ~~with arg~~ as array

Classes

→ Blueprint of object
→ Encapsulate data & function
→ support inheritance using super keyword
→ can used to implement interface

(class.ts)

class Quadrangle {

(See code)

//Property

side1 : number ; side2 : number ; side3 : number

side1: num

Constructor (param) {

},

& function

Constructor.

⇒ function keyword not important in class to initialise function

`let a = new className()` → new keyword used to create new object

⇒ If you not specify ~~class~~ access specifier it is by default public.

⇒ In inheritance child can use public or protected property → For inheritance

↑ can accessed out side class

↓ can't accessed outside class

See classes2.ts

⇒ Also you can able to create property inside constructor

Ex

```
class ~~square~~ rectangle {
    constructor (protected len: number, protected bre: number) {}
        ~~this.len = len; this.bre = bre~~ → so in turn this code not needed
}
```

⇒ creating property directly inside constructor

⇒ extend keyword used to do inheritance

⇒ child constructor must have super() function else throw error.

⇒ Also class has get & set which can used to set get and set property inside class

get nameset()

⇒ not much used

⇒ You can use get functions directly like prop

m.nameset not () ⇒ for get

() ⇒ for set

```
get nameret() {
    return this.nam
}
```

⇒ m = obj

Console.log(m.nameret)

↑ () not needed

Interfaces

→ A way of defining contract
→ interface can extend another interface
→ can set properties optional or readonly
→ can used to enforce contract on classes

can create interface

```
interface Person {
    name: string,
    mis : number
}
```

create object

```
const obj : Person = {
    name : "grs"
    mis : 1119o303
};
```

→ If interfaces have some undeclared functions

```
wait interface Hell {
    walk() : void;
}
```

→ no {}.

so the subsequent class or object created from that interface must have to implement that functions

⊛ See by doing

⇒ to make any property optional

`Property ? : type` → such code is used.

⇒ to make a property readonly

`readonly property : Datatype`

Such like that you can be able to crete a readonly property

If you want to create a class on top of interface we use imple

class Body implement interface{

—11—

}

here you have to implement not implemented functn in interface

⇒ contract in a sense interface force it's subsequent class, object to ~~cod~~ implement functn.

---

the interface block of code is not compiled by typescript compiler.

---

arrow funch abcc)

let abc= (param) => {

code

}

⇒ smaller & simple way to define functn

⇒ we can store functnon in variable & use it as funch

⇒ these are like lambda functns in Python

↑ But fat functns ~~are~~ can have multiline code inside functn.

```
let a = function (Param) {
        Code
    }
```

() => { } → multiline

(a) => console.log(a) → one line arrow fn

filter

L = [10, 20, 30, 40]

L.filter (func(item))

↑ this will use as

func → is function to print even or odd

↑ o/p => true / false

Modules

import & export used to use modules in Javascript

=> we can do it for function object

=> In javascript one file corrosponds to one module

=> export keyword used to export things from one file to another

you can import (ts) code in another (ts) file

=> you can export class, interface, function, const anything just use export keyword before it.

=> use vscode it make easy to import export.
(Auto import)

=> you can use (alias) if you have same names from diffent module using (as) keyword

If you want to export in one go

export { all file name, function name, class }

#Fetch Fetch api

Before Fetch XMLHttp is there which is hard

-> Fetch(url). then(responce => responce.json)
. then(function (data) {
code to manage & do stuff with data
})

=> Fetch is useful for many purposes

Fetch is far better than XMLHTTPrequest

-> you get data block as
. then (user => console.log(user))

=> you can get user or data it also depends on data which website is sending

#async & await

(see code)

async function is one function which use await keyword inside

return responce.json()
↑ It is a promise

=> If we want data so we have to return promise ie responce.json else our request will not get fullfilled.