Typescript

Typescript does static checking.

Typescript code is transpiled into javascript.
It is a development tool (because our project still runs JS)

- main.ts

```
let numOne = 3
let numTwo = "3"
let sum = numOne + numTwo
console.log(sum)
```

This is allowed.
→ prints 33.

So we will learn how we can enhance more of typesafety in our JS code using typescript.

- To convert .ts file into .js file, we use the command
  tsc intro.ts
  ↓
  filename

  . Also before that we need to install typescript globally.

Types:

Number, string, Boolean, Null, Undefined, Void, Object, Array, Tuples, Any, Never, unknown

Syntax:
let variableName : type = value

e.g;
let name : string = "Aman" | name. toUpperCase() ;

→ using dot gives suggestions for only string methods

- Primitives : string, number, and boolean

  number → JS does not have a special runtime value for integers, so there's no equivalent to int or float,

- Note : whenever we declare and initialise any variable at the same type. It is not recommended to declare type here,

```
let num: number = 40.2   ✗
let num = 40.2   ✓
```

• Any → whenever we don't want a particular value to cause typechecking errors.

We usually want to avoid this because any isn't type-checked.
Use the compiler flag noImplicitAny to flag any implicit any as an error

functions in typescript

Return type of function

passing default values.

Syntax:

```
function fname (name: string, email: string, isPaid: boolean = false):
{   let x: number;
    return x;
}
//Arrow function
const getHello = (s: string): string => {   }
```

//Map functions

```
const heros = ["thor", "spiderman", "ironman"]
heros.map((hero): string => {    Return type
    return `hero is ${hero}`;
})
```

// void keyword is used when a function does not return anything.

// never is used when a function doesn't return anything but throws an exception or terminates execution of the program.

```
function handleError (errmsg: string): never {
    throw new Error (errmsg);
}
```

Objects

• function createUser ({name: string, isPaid: boolean})
```
{ =
}
```

createUser ({name: "Aman", isPaid: false});

- when function returns an object.

```
function createCourse () : {name: string , price: number}
{
    return {name: "Reactjs", price: 399};
}
```

umber

- Odd behaviour of TS regarding objects

```
function createUser ({name: string, isPaid: boolean}) {}.
createUser ({ name: "Aman", isPaid: true }); ✓
createUser ({ name: "Aman", isPaid: true, email: "aman4u@gmail.com"});
                                            ✗ Compilation error.
let newUser = { name: "Aman", isPaid: true, email: "xyz@gmail.com" }.
createUser (newUser) ; ✓        This is odd behaviour, doesn't gives
                                error.
```

we dont use this type of parameters passed in function.

-

```
type User = {                         type aliases
    name : string;
    email: string;
    isActive: boolean
};

function createUser(user: User) {}
```

```
type User = {  ⟶  now once user created, then id cannot
                   be modified.
    readonly _id : string
    name : string
    email : string
    isActive : boolean
}   creditCard?: number
    └──────────→ ? symbol at last makes it optional.
```

- Combining two or more types

```
type cardNumber = {
        cardNumber: string
}
type cardDate = {
        carddate: string
}

type cardDetails = CardNumber & cardDate & {
        cvv: number
}
```

Arrays:

- Syntax:

```
const superHeros: string[] = [] ;
SuperHeros.push("spiderman");

const heroPower: number[] = []
            'or'
const heroPower: Array<number> = []
heroPower.push(2);

type User= {
        name: string
        isActive: boolean
}

Const allUsers: User[] = [].
```

- Defining 2-D arrays

```
const MLModels: number[][] = [ [255], [255, 16]].
```

Union

- let score: number | string = 33

  score = "55"

  or different type

- function returning multiple values, passing multiple data types

```
function getDbId (id: number | string)
{   id.toLowerCase();  ✗
    if( typeof id === "string")
    {
            id.toLowerCase();  ✓
    }.

    ... ☰
}
```

-
  ```
  const data : string [] | number [] = ["1", "hello"]; ✓
  const data2 : string [] | number [] = [1, 2]; ✓
  const data3 :     "       "     = [1, "hello"] ✗
  ```

  This can either be all numbers or all strings.

  ```
  const data4: (string | number) [] = [1, "hello"] ✓
  ```

  Here we can have both number & string.

- let ~~set~~ seatAllotment : "aisle" | "middle" | "window";

  This allows seatAllotment variable to take only three values

  ```
  seatAllotment = "aisle"; ✓
  seatAllotment = "crew"; ✗.
  ```

Tuples → tuples are a kind of array only with some restrictions, there datatype mentioned size are in order & fixed.

let rgb : [number, number, number] = [255, 123, 112] . ✓.

rgb = [255, 216, 93 , 109] X .

let tUser : [string, number, boolean] = ["hc", 131, true]; ✓.

tUser = ["hc", true, 121] X .

- odd behaviour.
  tUser [0] = "Hello" ; ✓ (values might be changed)
  tUser. push (true) ✓. (allowed) allowed to use methods of arrays.

  So be cautious about that

Enum

Syntax:

```
enum SeatChoice {
    AISLE,      → by default 0, next values are incremented by 1.
    MIDDLE,
    WINDOW,
}

const hcSeat = SeatChoice. AISLE;
```

Note: when compiled to JS , immediately executed function is generated. for this. This produces large code.

```
const enum SeatChoice {
    AISLE = "aisle",
    MIDDLE = 3,
    WINDOW
}
const hcSeat = SeatChoice .AISLE;
```

} Using const with enum produces less code.

Interfaces

Syntax:

(1)
```
interface User {
    readonly dbId: number,
    email : string,
    userId: number,
    googleId?: string,
    startTrail : () => string
            or
    startTrail () : string
}

const aman: User = { dbId: 22, email: "h@h.com", userId: 2211,
        startTrail : () => {
            return " trail started";
        }
}
```

- Reopening of the interface:

```
interface User {
    email: string
}
interface User{
    githubToken: string.
}

const aman: User = { email: "aman@gmail.com", githubtoken : "1defe"}.
```

- ~~Extends~~ ~~the~~ Inheritance

```
interface Admin extends User
{
    role : "admin" | "ta" ;
}
```

Using typescript in project

Step 1 - create a folder
Step 2 - tsc --init → this creates a typescript config file

Run command "tsc -w" to keep typescript file combining.
after setting the ~~opt~~ output folder.

Classes

```
class User {
    email: string
    name: string
    city: string = ""   → default value.

    constructor (email: string, name: string)
    { this. email = email;
       this. name = name;
    }
}.

const aman = new User ("a@gmail.com", "aman");
```

By default everything is public in classes

```
class User {
    private email: string
    name: string
    constructor (email: string, name: string)
    {  this. email = email;
       this. name = name;
    }
}.
```

• More precise way developers use:

```
class User {

    readonly city: string = "Delhi"
    constructor (
        public email: string,
        public name: string
        ){ ==
    }
}
```

- getter & setter methods.

```
  class User {
     private _courseCount = 1
     readonly city: string = "Jaipur"
     constructor (
          public email: string,
          public name: string,
        ){
     } =

     get getAppleEmail(): string {
          return `apple ${this.email}`
     }
     get courseCount(): number {
          return this._courseCount;
     }
```

getter methods

```
     set courseCount (courseNum) {
          this._courseCount = courseNum }.
     }
```

setter methods cannot have return type annotations like void, etc

Abstract classes in TS.

Abstract classes do not allow us to create object of their own, but help to define classes inheriting them. There methods can be abstract, can have definitions, can be overided. also

Generics

```
  function identityThree ~~f~~<Type> (val: Type): Type {
       return val;
  }

  function identityFour<T> (val: T): T {
       return val;
  }
```

example:

```
function getSearchProducts <T> (products: T[]): T {
        // do some database operations
        const myIndex = 3
        return products[myIndex]
}
```

'or'

~~const getMoreSearchProducts = <T>(): => { };~~

or <T,> to clarify that its generics not Template

```
const getSearchProducts = <T> (products: T[]): T => {
    return products[4];
}
```