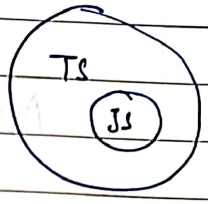
__/__/____

Typescript.

– (superscript of JS.)



Doesn't give you more features.
Allows to write JS in precise manner.
↳ To avoid Errors.

code in TS is compiled in JS.

→ TS is not about reinventing JS
(Write JS in precise manner)

→ should not use TS if a small project.

TS is All about Type safety.

JS allows to do odd behaviour
2 + "2" JS doesn't stop this.
'22'

But it should not be allowed
null + 2 undefined + 2
2. NaN

We haven't been concern about type.

What TS is not.
What TS does:
static checking ~JAVA
Parser of long checks code

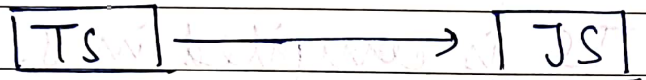
__/__/____

TS is just for static checking
(At runtime)

Analyse code as we type.

TS is code. X Not always sometimes more code than JS.

We write .ts.



transpited to

TS code is converted to JS.

→ TS is a devtool. Not a language.
Your project still runs on JS.
(TS: Wrapper around JS)

Squiggly line on errors

```
let user = {name: "Harsh", age: 10};
let email = user.email
```

Error in TS but Runs fine in JS.

```
let a = 3
    b = "3"
sum = a+b
```

} No error in both.

Global system install (system wide)
In React diff config (ts configfile)

> terminal.
npm i -g typescript. (Global)
--save-dev
(Projects)

Topic 1:- Basic Types & Variables

(1) Variable Declaration: var let const.
var - old way of declaring (avoid now)
let - prefer for var. that can change
const - prefer for constant

Rule:- Always use const when you try to change variable's value.

(2) Basic Data Types:-

- number: let a: number = 12, 12.3, 23444..
- string: let b: string = "hello"
- boolean: let isActive: boolean = true

(3) Primitive V/S Reference Types:-

- Primitive: number, string, boolean
- Reference: array, object, functions.

__/__/____

- **Primitive:-**
  Stored directly in memory wt value copy behaviour.

  Number, string, boolean.

```
let a = 15
let b = a
b += 2
log(a)   15
log(b)   17
```

copy of 'a' assigned to 'b' only.

- **Reference:-**
  stored by reference. changes to one will affect original.

  Array, object, fn

```
let a = [1,2,3,4]
let b = a
b.pop()
log(a)  [1,2,3]
```

'b' refers to same array as 'a' same memory location.

(4) Type Inference

TS can automatically detect variable type based on assigned value.

```
let myNum = 5; // inferred as number.
myNum.toUppercase(); // ISN'T A METHOD ON NUMBER.
```

__/__/__

- Prevents accidental misuse by automatically identifying variable's type
  - Say if greetings is a string.
    will suggest appropriate string methods only.
    .toLowerCase() etc.

(5) Type Annotations.

TS can infer types but explicitly specifying types
- improve code clarity
- minimise Errors.

```
let greet: string;  // Explicit Type Annotation.
greet = "Hello";
```

Tip → for simple cases rely on inference
But use Type Annotation for complex data structures.

SUMMARY

~> TypeScript Intellisense & Type-checking features improves code quality & catch errors.

~~MID-SEM~~

~> Prefer let, const over var
~> Diff between primitive & reference types
~> use TS Inference in simple cases, annotations in complex.

MOCK Ques:-

1) `var x = 10;`
`if (true) {`
`var x = 20;`
`}`
`log(x)`

O/P = 20.

var is f^n scope NOT Block scope

→ Refuse to same x declared

use let `let x = 10` & FIX.

2) `let num = 42`
`num = "hello"`

string not assignable to type 'number'

Type Influence

FIX

`let num: string | number;`

UNION.

3) `.toLowerCase()` ✓
`.toLowercase();` ✗

Type mistake

CASE-SENSITIVE.

4) `let arr1 = [10, 20, 30]`
`let arr2 = arr1`
`arr2.push(100);`
`log(arr1)` [10, 20, 30, 100]
`(arr2)` [10, 20, 30, 100]

REFER TO SAME MEMORY.

→ CREATE COPY instead of REFERENCE:-

`let arr2 = [...arr1];`

__/__/__

5) let a = [1, 2, 3, 4]
Without modifying create 'b' which does not have last element
let b = a.slice(0, -1); // [1, 2, 3]

creates new array w/o modifying original.

6) let values = [1, "two", true];
.push(false) ✓
YES.
TS infers it as Array<number | string | boolean>.
let values: number[] = [1, 2, 3].

- Type ANY
flexible type
can Hold any value { string, number, object etc.
Disables TS type-checking for that particular value.
RISKY
let data: any = 42;
data.toUpperCase(); } Runtime TS won't

Use case:-
For API when we don't know catch this.

→ Use unknown:-
let data: unknown = "Hello";
if (typeof data === "string") {
data.toUpperCase();
}

__/__/____

TOPIC 3: functions

F^n's are powerful → can be ~~the~~ enhanced using Type Annotations to ensure better type safety & improved code.

1) Defining

By default - Treated as Any
Lead to unexpected behav.

```
function addTwo(num) {
    return num + 2;
}

addTwo(5)    7
("5")  52
Problems
```

```
function addTwo
(addParams: number)
    : number {

}
```

: number ← return type

Parameter type.

```
function getUpper(val: string): string {
    return val.toUpperCase();
}
```

Multiple Params / void

```
function signUp
(name: string, email: string, isPaid:
boolean)
    : void {
    .log(name, email, isPaid).
```

__/__/____

signupUser("Harsh", "h@mail.com");

missing isPaid Parameter.

3) Default Parameters.

function loginUser
(name: string, email: string, isPaid=false)
{

}

Default.

use Explicit return-types whenever possible

function addThree(num: number){
~~log~~ return "Hello"
}

FIX. → ~~: string~~ : number.

4) union Types in f"s.
multiple return types

function getVal(myVal: number)
: boolean | string {
if (myVal > 5)
return true
return "okay"
}

__/__/____

5) Arrow fn wt Return Types.

```
const getHello = (s: string): string => {
    return `Hello ${s}`;
};
```

6) Map

TS can automatically detect type in .map() methods.

```
const heroes = ["Thor", "Spidy"...];
heroes.map((hero) => {
    return `Hero is ${hero}`;
});
```

```
heroes.map(hero): string => {
    return `Hero is ${hero}`;
});
```

7) Fn wt never

fn that never returns — fn throwing errors or ∞ loop.

__/__/____

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

never ensures that a fn never reaches return statement.

factorial :-

```
function fact(n: number): number {
    if (n < 0)
        throw new Error("-ve val");

    if (n == = = 0)
        return 1

    return n * fact(n-1);
}
```

Returning Array

```
function formatVal(values: number[])
    : string[] {
    values.map((value): number => {
        return `value is ${value}`;
    });
}
```

__/__/____

TOPIC 4:- Bad Behavior Objects.

↳ Objects, Type Aliases, Readonly.

1 → Objects

↳ Powerful way to structure & manage data.

Passed as fn param, returned from fn, defined wt specific types for type safety.

```
const User = {
    name: "haush",
    email: "h@mail.com",
    isActive: true.
}

function createUser
({ name, email }: { name: string,
                    email: string }) {

    log ("user");
}.

createUser(
  {
      name: "Haush",
      email: "h@mail.com"
  }
)
```

But Extra Params will cause error.

__/__/____

```
createUser(
  { name:
    email:
    isPaid:
  }
)
```

Extra prop will cause error.

createUser(user); Correct way to handle.

When we pass an object literal to fn, TS performs strict security checks.
TS assumes it to be mistake & warns.
createUser expects → 2 Properties (name, Email) IsPaid will cause error.

But if you store in object & call, in variable first
TS performs less checking as it might be used somewhere else in code.
Does not restrict extra properties.

Through email & name will be accessible & isPaid will be ignored.

TO pass direct Object Literal → Index Signature

```
function createUser
(user: {name: string, isPaid: boolean,
        [key: string]: any})
```

→ Object as return type:-

```
function createCourse(): {name: string, price: number} {
    return {
        name: "React",
        price: 500
    }
};
```

(2) Type Aliases.

Simplify complex type definitions by creating reusable custom-types.

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

function createUser(user: User): User {
    return {
        name: "h"
        email: "h@com"
        isPaid: true.
    };
};
```

✓ Reduces redundant code
✓ Improves readability.
✓ Helps to define complex D.S.

→ Read Only

```
type User = {
    readonly id : string;
    name : string;
}
```

(1)

```
let myUser : User = {
    id : 123,
    name : "Vansh"
}
```

myUser.name = "Vansh" ✓
myUser.id = 567 ✗

Ensures certain object properties cannot be changed after initialisation.

optional prop

at (1)

```
creditDetail ?: number;
```

↳ optional property. Allow flexibility.

✓ Readonly is ideal for IDs that shouldn't change
✓ '?' provides flexibility in D.S.

→ Intersection : Merges multiple Types.

```
type cardNumber = {
    cardNum: string
}
type cardExp = {
    cardexpiry: string.
}

type cardDetails = cardNumber & cardExp & {
    cardCVV: number
}

const mycard : cardDetails = {
    cardNum: 12345678
    cardExp: 12/25
    cardCVV: 456
}
```

useful for combining data from multiple sources.

PROBLEMS.

1) Opt object Params

```
function create(
    user : {name: string, age?: number})
```

2)

Type	V/S	Object.
Describes Blueprint		Represents actual object stored in memory.
Exists at compile time		created at runtime
supports union, intersection etc.		cannot redefine.
Ensures structure/ type of data		Does not enforce unless specified

3) type Status = "success" | "fail" | "error";

function getstatus(status : Status) {

}.

__/__/____

TOPICS :- Arrays, Unions & Tuples.

ARRAY

↳ collection of elements that share same data type.
↳ TS enforces strict type checking in arrays.

1) std syntax

```
const superHeros: number[] = [];
superHeroes.push(2);
```

2) using Array<>

```
const getPower: Array<number> = [];
getPower.push(5);
```

3) using Type

```
type People = {
    name: String,
    isActive: boolean,
}

const allPeople: People[] = [];

allPeople.push({
    name: "Harsh",
    isActive: false,
});
```

__/__/__

4) 2D Arrays

```
const MLModel: number[][] = [
        [255, 255, 255],
        [0, 0, 0]
];
```

5) Readonly Array

Ensures array is immutable. You cannot modify its elements.

```
const numbers: ReadonlyArray<number>
                = [1, 2, 3];

numbers.push(10); // NOT ALLOWED.
```

Does not exist on Readonly[].

UNION

Allows variable/parameter to accept more than one data type.
Safer & Better than ANY.

→ union with variables:-

```
let score: number | string | boolean = 45;
score = "Pass"
score = false.
```

→ union of Objects.

→ Union with Objects.

```
type usr = { name: string, id: number }
type admin = { admin: string, id: number };

let Harsh: usr | admin
    = { name: "Harsh", id: "123" }.
Harsh = { admin: "Hari", id = "000" }.
```

Now acting as admin.

→ Union with functions.

```
function getDBId ( id: number | string) {
  if (typeof id === "string") {
    } Treated as string.
  } else {
    } Treated as number.
  }
```

→ Union with Arrays.

(1) Either number or strings (NOT MIXED.)

```
const data: number[] | string[]
    = [1, 2, 3].
```

Either all num, or all ans.

__/__/____

(2) Mixed Data within single array.

```
const data: (number | string)[]
    = ["one", 2, "three"].
```

TUPLES

Special Array which has fixed order & specific data type for its elements.

Fixed length. (By default)

```
let user : [string, number, boolean];
user = ["harsh", 123, true];
```

FIXED ORDER, FIXED TYPE.

```
user.push(5)   // ERROR.
```

✓

Tuples specify exact no of elements & types.
Each index corresponds to exact type.
↳ ensuring strict checking.

Flexible length tuple:-

```
let nums: [number, ...string[]]
nums = [1, "one", "two", "three"];
```

__/__/____

→ Tuple wit type alias

type User = [string, number]

let newUser: User = ["123", 867]

newUser[0] = "hello" } Allowed

newUser.pop()
.push(true) } NOT ALLOWED.

→ Tuple wit Fixed length

Eg:- RGB values

let Rgb: [number, number, number]
= [255, 135, 100];

LITERAL TYPES

↳ specific value that acts type itself.

let pi: 3.14 = 3.14 // Fixed value
pi = 4.13 X

let seatAllot: "aisle" | "middle" | "window";
seatAllot = "crew" X

Not a part of defined types.

__/__/____

TOPIC 6 :- ENUMS, INTERFACES.

(1) ENUMS

→ Provide a way to define named constants.

→ Improves readability, ensures that values remain constant.

```
enum SeatChoice {
    aisle,
    middle,
    window
};

const mySeat = SeatChoice.aisle;
```

O/P ⇒ 0.

Assign value 0,
Enum numbers 0 from default.
(0, 1, 2)

```
// custom values
enum SeatChoice {
    aisle = 10,
    middle = 23,
    window
};
```

(24) Auto increments.

__/__/____

```
enum SeatChoice {
    aisle = "aisle",
    window = "window"
};
```

String values.

All must be explicitly assigned

1. For Efficiency use

```
const enum SeatChoice {
    aisle = "aisle",
    window = "window"
};
```

→ Reduces JS code → Improving Performance.

2. Interfaces in Typescript.

↳ Used to define structure of object. Ensures it follows specific shape.

```
interface NewUser {
    readonly dbId: number,
    email: string,
    phone?: number // optional.

    startTrail(): string
    getCoupon(param: string): number.
};
```

const alpha : NewUser = {
dbId: 22,
email: "n@mail.com",
getCoupon: (name: "Harish") => 10

→ REOPENING in Interfaces.

↳ can be reopened & extended by adding new properties.

interface NewUser {
gitHub: string.

Now the object has been added with new property ⟼ github.

→ Interface Inheritance.

Can extend other Interfaces to include add. properties.

interface Admin extends NewUser {
role: "admin" | "ta" | "learner".
}

extends I_1, I_2, I_3
multiple inherit

__/__/____

→ Interface V/S Type.

Interface	Type.
supports extending with extends.	supports extending with intersections.
can be reopened.	cannot be reopened.
suitable for object definitions	suitable for both objects & primitive.

type Person = string | boolean

→ combined usage.

```
interface Vehicle {
    brand: string
    speed: number
}

type Fuel = "Petrol" | "Diesel"
            Type

interface car extends Vehicle {
    fuel: FuelType;
}.
```

__/__/____

TOPIC 7:- classes in TS.

(1) classes in TS:-

Blueprint for creating objects.
Methods to describe the behaviour.

JS class:-

```
class UserPerson {
    constructor(email, name) {
        this.email = email
        this.name = name.
    }
}.
```

TS class:-

```
class UserPerson {
    email : string
    name : string
    city : string = "";  // Default.

    readonly country : string = "India";

                    Readonly Prop.

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

__/__/____

(2) Access Modifier.

public — Access anywhere (DEFAULT)

private — only inside class

protected — Accessible inside the class and inherited classes.

```
class User {
    public email: string
    public name: string
    private readonly city: string = "Jaipur";
    constructor(email: string, name: string) {
        this.email = email
        this.name = name
        this.city = "Delhi"
    }
}
```

{ can be modified inside the class.

```
const user = new User("naush", "n@mail.com")
```

(3) Get / set → used to modify private/protected

→ used to access private/protected.

* Setters don't have return type.

__/__/____

```
class User {
    private courseCount = 1;

    get courseCnt(): number {
        return this.courseCount;
    }

    set courseCntSet(param: number) {
        this.courseCount = param;
    }
}

const person = new User();

person.courseCntSet(5)   // This is not a fn.

person.courseCntSet = 5;
```

This is treated as a property instead of method.

get ⟶ Reading
set ⟶ Assigning a value
} To property

```
log(person.courseCntSet) // 5.
```

(4) Inheritance

private → Not inherited
protected → Inherited.

```
class Parent {
    protected count = 1;

    getCourseCnt(): number {
        return this.count;
    }
}

class Child extends Parent {

    changeCnt() {
        this.count = 5;
    }
}
```

Child → Obj

```
const subUser = new Child();
subUser.changeCnt();
log(subUser.getCourseCnt); // 5

const parentUser = new Parent();
log(parentUser.getCourseCnt); // 1
```

INSTANCE specific behaviour.
Own copy.

__/__/____

(5) Implementing Interfaces

↓

Define structure a class must follow.

```
interface takePhoto {
    cameraMode: string
    filter: string
}

class Instagram implements takePhoto {   (1)
    constructor (
        public cameraMode: string,
        public filter : string.
    ) {


    }
}
```

(2)

Multiple interfaces using ;

say

```
interface takeVideo {
    record : startRecord();
}
```

At (1) add , takeVideo

Also define all methods and properties.

At (2)

```
startRecord(): void {
    log("create");
```

__/__/____

console.log(obj.createStory()) logged -

O/P

story was created
undefined.

bz nothing is being returned.