TYPESCRIPT

- Created at Microsoft back in 2012.
- It was created to address some short comings of JavaScript.

JavaScript is like a kid without and discipline and does everything what he wants on the other hand **TypeScript** is like a kid with some discipline.

Benefits:

- **Type safety:** Catches errors before running.
- TypeScript always support latest ECMAScript features.
- **Scalable code:** Easier to maintain on big projects.

Drawbacks:

- There is always a compilation process because browsers don't understand TS yet.

Note: Conversion of **TypeScript** code into **JavaScript** code is called **Transpilation**.

DATA TYPES IN TYPESCRIPT

Primitive types:

- number
- string
- boolean
- null
- undefined
- bigInt
- symbol

Special types:

- any:
- unknown
- void
- never

Complex types:

- object
- array
- tuple
- enum

Any: disables type checking. The variable can hold **any** value, like plain JavaScript.

```
let level: any;
level = 1;
level = "2"
```

Arrays: hold values of a specific type. Two ways:

```
let nums: number[] = [1, 2, 3];        // Type[] syntax
let names: Array<string> = ["a", "b"]; // Generic syntax
```

Tuples: fixed-length arrays with known types at each position.

```
let user: [string, number] = ["Alice", 25];
// string at 0, number at 1
```

Enums: defines named constants for better readability. By default, values are numbers starting at 0.

```
enum Direction {Up, Down, Left, Right} // Value of Up is 0, Down 1 and so on.
enum Direction { Up = "u", Down = "d", Left = "l", Right = "r" }

let move: Direction = Direction.Up;
```

Objects: TypeScript need type definitions for properties Can also use type (also known as type alias) or interface for reuse.

```
let user: {
    name: string;
    age: number
} = {
    name: "Alice",
    age: 25
};
```

Optional & Read-only Properties:

```
type User = {
    readonly id: number;
    name: string;
    age?: number; // optional
};

let user2: User = { id: 1, name: "Alice" };
user2.name = "Bob";   // OK
// user.id = 2;      // Error: readonly
```

Unknown: means “type not known yet,” but safer than any because you **must check** before using it.

```
let data: unknown;

data = "hello";
// console.log(data.toUpperCase()); // Error: unknown

if (typeof data === "string") {
    console.log(data.toUpperCase()); // OK after check
}
```

Unknown Vs Any:

- any = no type safety, TS stops checking.
- unknown = forces type checks before use.

```
let a: any = "hello";
a.toUpperCase(); // OK, no checks → unsafe

let u: unknown = "hello";
// u.toUpperCase(); // Error, must check first → safer
if (typeof u === "string") {
    u.toUpperCase(); // OK after narrowing
}
```

Never: never means a value never happens. Used for:

- **Functions that never return** (e.g., throw errors):
- **Exhaustive checks** in unions:

```
function fail(msg: string): never {
    throw new Error(msg);
}

type Shape = "circle" | "square";
function area(shape: Shape) {
    if (shape === "circle") return 1;
    if (shape === "square") return 2;
    const _exhaustive: never = shape; // error if new type added
}

// If you later add "triangle" to Shape, TypeScript forces you to handle it.
```

NOTE: Exhaustive means **covering all possible cases** so nothing is missed. In TypeScript, used with union types to ensure every option is handled.

Union Types & Narrowing:

Union types let a variable hold **more than one type**:

Narrowing means refining a **broad type** to a **specific type** at runtime using checks.

```
function printId(id: number | string) {
    // Narrowing
    if (typeof id === "string") {
        console.log(id.toUpperCase()); // string methods
    } else {
        console.log(id.toFixed(2));    // number methods
    }
}
```

Intersection Types: combine multiple types into one. Object must satisfy all combined types.

```
type Dragable = {
    drag: () => void
}

type Resizeable = {
    resize: () => void
}

type UIWidget = Dragable & Resizeable;

let TextBox: UIWidget = {
    drag: () => { },
    resize: () => { }
}
```

Type Literals: Type literals are exact values as types, not just general types.

```
let direction: "up" | "down";
direction = "up";   // OK
direction = "left"; // Error
```

Nullable Types:

Typescript is very strict with null and undefined as they are come source of bugs in our app. Nullable types allow null or undefined with another type using union. But what if a value can be null.

```
function greet(name: string): void {
  console.log(`Hey ${name.toUpperCase()}`); // toUpperCase crashes on null
}
greet(null); // Error


function greet(name: string | null | undefined): void {
    if (name)
        console.log(`Hey! ${name.toUpperCase()}`);
    else
        console.log("Hey!");
}
greet(null); // No Error
```

Nullish Coalescing Operator:

TypeScript and JavaScript have following falsie values:

False, 0, -0, 0n (BigInt zero) "" (empty string) null undefined NaN

Everything else is truthy.

Means some if statement will never enter the if body for these values like:

```
if (variable_name)
      // NEVER RUNS FOR ALL FALSY VALUES
```

Now the problem is **0** or **"" string** may possibly a truthy value e.g. Speed of car can be 0. For this purpose, we have to use if checks to specially tackle undefined and null like:

```
  if(variable !== null && variable !== undefined)
```

But we have a more optimized way to do it.

Here comes the **Nullish Coaelscing Operator "??"**. The **"??"** operator only checks for null and undefined.

```
let count = 0;
let result = count ?? 10; // result will be 10 only if count is undefined or null
```

Type Assertion: tells TypeScript, “Trust me, I know this type.”

```
const input = document.getElementById("username") as HTMLInputElement;
input.value = "Ismail"; // TS knows it's an input, not just HTMLElement
```

Without assertion, TS only knows **getElementById** returns **HTMLElement** or **null**, so .**value** would error.

Two Syntaxes:

```
let value: unknown = "hello";

// angle-bracket
let len1 = (<string>value).length;

// as keyword
let len2 = (value as string).length;
```

FUNCTIONS

Functions in TypeScript need parameter and return types:

```
function add(a: number, b: number): number {
    return a + b;
}

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

// Optional Parameter
function greet(name?: string): void {
    console.log(`Hello ${name || "Guest"}`);
}

// Default Parameter
function greet2(name: string = "Guest"): void {
    console.log(`Hello ${name}`);
}

// Rest Parameter
function sum(...nums: number[]): number {
    return nums.reduce((a, b) => a + b, 0);
}
sum(1, 2, 3, 4); // 10
```

Rest parameter collects **all remaining arguments** into an array for flexible function calls.

OPP IN TYPESCRIPT

- **Class**: Blueprint for objects.
- **Object**: Instance of a class.
- **Encapsulation**: public, private, protected control access.
- **Inheritance**: extends keyword shares properties/methods.
- **Polymorphism**: Override methods in child classes.
- **Abstraction**: Abstract classes/interfaces for structure only.

```
class Account {
    //  ATTRIBUTES / PROPERTIES
    id: number;
    owner: string;
    balance: number;

    // ACTIONS / METHODS / FUNCTIONS / PROCEDURES
    deposit(amount: number) {
        if (amount <= 0) {
            throw new Error("Invalid Amount")
        }
        this.balance += amount;
    }
}
```

Constructor: is a special method that runs when you create a class object. It initializes properties.

```
    constructor(id: number, owner: string, balance: number) {
        this.id = id;
        this.owner = owner
        this.balance = balance;
    };
```

Type & Instance of Class:

```
let acc = new Account(1, "Ismail", 0);
console.log(typeof acc); // OBJECT

// FOR EVERY CLASS THE typeof IS OBJECT
// NOW WHAT IF WE WANT TO CHECK THAT acc BELONGS TO Account OR NOT
// WE CAN USE instanceof

console.log(acc instanceof Account); // TRUE
```

Read-Only & Optional Properties:

```
class Account {
    readonly id: number; // READONLY
    owner: string;
    balance: number;
    type?: string // OPTIONAL
}
```

Access Modifiers:

- **Public**: default, accessible anywhere.
- **Private**: only inside the same class.
- **Protected**: inside the same class + subclasses.

Parameter Properties: let you declare and initialize class properties right in the constructor using access modifiers.

```
// WITHOUT PARAMTER PROPERTIES
class Account {
    readonly id: number; // READONLY
    owner: string;
    balance: number;
    type?: string // OPTIONAL

    constructor(id: number, owner: string, balance: number, type: string) {
        this.id = id;
        this.owner = owner
        this.balance = balance;
        this.type = type;
    };
}

// WITH PARAMTER PROPERTIES
class Account {
    constructor(
        public readonly id: number,
        public owner: string,
        public balance: number,
        public type?: string
    ) { };
}
```

Getters and Setters: control how properties are read or updated.

```
class Account {
    constructor(private _balance: number) { }

    get balance() {
        return this._balance; // read
    }

    set balance(amount: number) {
        if (amount >= 0) this._balance = amount; // write with check
    }
}

const acc = new Account(100);
console.log(acc.balance); // uses getter
acc.balance = 200;        // uses setter
```

The _ prefix is just a **convention**, not required. Used to:

- Mark private fields clearly.
- Avoid name clash with getter/setter names.

Index Signatures: let you define the type of dynamic property names in an object.

```
type PhoneBook = {
    [name: string]: string; // key: string, value: string
};

let contacts: PhoneBook = {
    Alice: "1234",
    Bob: "5678"
};

contacts.Ismail = "9101"
```

Static Members: are used when data or behavior belongs to the **class as a whole**, not to each object. Static members belong to the class itself, not to instances.

```
class Ride {
    static activeRides: number = 0;

    start() { Ride.activeRides++; }

    stop() { Ride.activeRides--; }
}
```

```
let ride1 = new Ride();
ride1.start();

let ride2 = new Ride();
ride2.start();

console.log(Ride.activeRides); // Class name to access not obj name
```

Inheritance: lets a class reuse another class's properties and methods using extends.

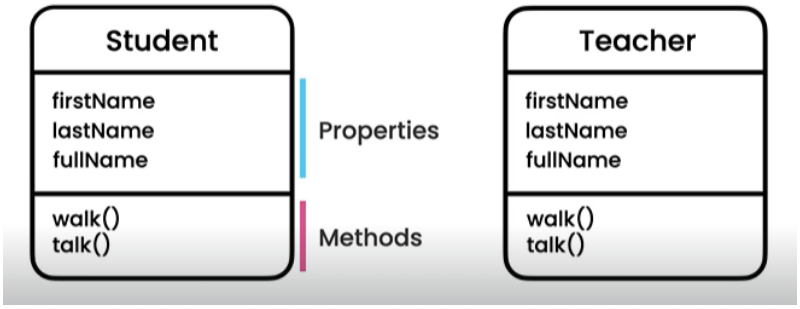

Figure 1: Similar Attributes & Methods in 2 Diff Classes

```
class Person {
    constructor(
        public firstName: string,
        public lastName: string
    ) { }

    get fullName() { return `${this.firstName} ${this.lastName}`; }

    walk() { console.log(`${this.fullName} is walking`); }
    talk() { console.log(`${this.fullName} is talking`); }
}

class Student extends Person {
    constructor(
        public studentId: number,
        public firstName: string,
        public lastName: string
    ) {
        super(firstName, lastName)
    }

    study() { console.log(`${this.fullName} is studying`); }
}

class Teacher extends Person {
    constructor(
        public teacherId: number,
```

```
        public firstName: string,
        public lastName: string
    ) {
        super(firstName, lastName)
    }
    teach() { console.log(`${this.fullName} is teaching`); }
}

const s = new Student(1, "Alice", "Smith");
s.walk();
s.study();

const t = new Teacher(1, "Bob", "Jones");
t.talk();
t.teach();
```

Method Overriding: In inheritance, **overriding** means a child class provides its own version of a parent method.

```
// We want to prefix 'Prof' for every teacher
class Teacher extends Person {
    override get fullName(): string {
        return "Prof" + super.fullName;
    }
}
```

Polymorphism:

Polymorphism = "many forms."

It means different classes can be treated as the same type (usually the parent), but they behave differently when you call their methods.

```
// Polymorphism in action
const people: Person[] = [
    new Person("Alex", "Hales"),
    new Student(1, "Alice", ""),
    new Teacher("Bob", "Doe")
];

for (let person of people) {
    console.log(person.fullName);
}
```

Abstract Classes and Methods:

Abstracts are like blueprints.

- **Abstract class** = can't be directly instantiated. Used only as a base.
- **Abstract method** = must be implemented in child classes.

```
abstract class Shape {
    constructor(public color: string) { }

    // abstract method → no body, must be implemented
    abstract getArea(): number;

    // normal method → shared logic
    describe() {
        return `A ${this.color} shape`;
    }
}

class Circle extends Shape {
    constructor(color: string, public radius: number) {
        super(color);
    }

    override getArea(): number {
        return Math.PI * this.radius * this.radius;
    }
}

class Square extends Shape {
    constructor(color: string, public side: number) {
        super(color);
    }

    override getArea(): number {
        return this.side * this.side;
    }
}

const shapes: Shape[] = [
    new Circle("red", 5),
    new Square("blue", 4)
];

for (let s of shapes) {
    console.log(s.describe(), "Area:", s.getArea());
}
```

Interfaces: **Interface** = contract that defines the *shape* of an object or class. It tells TypeScript "This thing must look like this." No implementation, just structure.

Key points:

- Interfaces define **what** must exist, not **how**.
- A class or object must satisfy all the fields and methods an interface requires.
- Interfaces support inheritance (extends) and multiple inheritance (a class can implement many interfaces).

GENERICS

Generics = write code once, reuse with many types.

They let you create reusable, type-safe components without locking to a specific type.

```
// EXAMPLE WITH A FUNCTION
function identity<T>(value: T): T {
    return value;
}

let num = identity<number>(42);   // T = number
let str = identity<string>("Hi"); // T = string


// EXAMPLE WITH A CLASS
class Box<T> {
    constructor(public content: T) { }
}

const numBox = new Box<number>(123);
const strBox = new Box<string>("hello");


// EXAMPLE WITH ARRAYS
function getFirst<T>(arr: T[]): T {
    return arr[0];
}

console.log(getFirst([1, 2, 3]));       // number
console.log(getFirst(["a", "b", "c"])); // string
```

Generics = **placeholders for types**. They give flexibility while still keeping type safety.

Constraints in generics:

Constraints in generics let you limit what types are allowed.

```
// EXAMPLE: ONLY NUMBERS OR STRINGS ALLOWED
function echoId<T extends number | string>(id: T): T {
    return id;
}

echoId(123);     // ok
echoId("abc");   // ok
echoId(true); // ❌ error


// EXAMPLE: MUST HAVE A CERTAIN PROPERTY
interface HasLength {
    length: number;
}

function logLength<T extends HasLength>(item: T): void {
    console.log(item.length);
}

logLength("hello");      // ok (string has length)
logLength([1, 2, 3]);    // ok (array has length)
logLength(42);        // error (number has no length)


// EXAMPLE: CLASS WITH CONSTRAINT
class Repository<T extends { id: number }> {
    private items: T[] = [];

    add(item: T) {
        this.items.push(item);
    }
}

const repo = new Repository<{ id: number; name: string }>();
repo.add({ id: 1, name: "Alice" });
```

- Without constraints = very flexible but unsafe.
- With constraints = still generic, but controlled.

Key Of Operator:

The keyof operator in TypeScript creates a **union of all property names (keys)** of a type.

Example:

```
type Person = {
    id: number;
    name: string;
    age: number;
};

type PersonKeys = keyof Person; // "id" | "name" | "age"
```

So **PersonKeys** can only be **"id" | "name" | "age".**

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];
}

const person: Person = { id: 1, name: "Alice", age: 25 };

let nameValue = getProperty(person, "name"); // string
// getProperty(person, "address"); error, address is not a key
```

Type Mapping:

Think of it as **a for-loop for types**.

You take each key of a type and transform it.

Normal type:

```
type Person = {
    id: number;
    name: string;
    age: number;
};
```

Type mapping:

```
// loop over each key (id, name, age)
type OptionalPerson = {
    [K in keyof Person]?: Person[K];
};
```

That turns into:

```
type OptionalPerson = {
    id?: number;
    name?: string;
    age?: number;
};
```

Another Example:

```
type ReadonlyPerson = {
    [K in keyof Person]: Readonly<Person[K]>;
};
```

That becomes:

```
type ReadonlyPerson = {
    readonly id: number;
    readonly name: string;
    readonly age: number;
};
```

So:

- keyof Person = "id" | "name" | “age
- [K in keyof Person] = loop over "id", "name" and “age”
- Person[K] = type of that property

It’s just a way to **copy a type and tweak its properties** without rewriting.

DECORATORS

Decorators in TypeScript = **special functions** that can attach extra behavior to classes, methods, properties, or parameters.
They look like @something placed above a definition.

Note: They are still **experimental** (need "experimentalDecorators": true in tsconfig.json).

```
// 1. Class Decorator
function Logger(constructor: Function) {
    console.log("Class created!");
}
@Logger
```

```
class Person { } // Just prints "Class created!" when the class is defined.


// 2. Method Decorator
function Log(target: any, key: string, desc: PropertyDescriptor) {
    console.log("Method:", key);
}
class Calculator {
    @Log // Prints "Method: add" when class is loaded.
    add(a: number, b: number) {
        return a + b;
    }
}


// 3. Property Decorator
function Prop(target: any, key: string) {
    console.log("Property:", key);
}
class Car {
    @Prop // Prints "Property: model".
    model: string = "Tesla";
}


// 4. Parameter Decorator
function Param(target: any, key: string, index: number) {
    console.log("Parameter at position:", index);
}
class User {
    greet(@Param message: string) { } // Prints "Parameter at position: 0".
}
```

In simple words: **decorators are like wrappers** you stick on classes or methods to add logging, validation, authorization, etc., without touching the core code.

In one line:

@ **before class/method/property/parameter** = attach a function that runs when the class is created.

Note: While calling more than one decorator to a **class/method/property/parameter** the decorators are called in reverse order **(the last decorator will be called first)**