

TYPESCRIPT

Npm i typescript -g

Npx tsc - - init → create a tsconfig.json

Tsc [app.js](#)

Tsc - - watch

Primitives & Reference Types in TypeScript

Primitive Types (Prakar ke mool type):

Primitive types woh hote hain jo by value store hote hain — iska matlab hai inka ek copy banti hai jab hum inhe kisi variable mein assign karte hain.

Ye immutable hote hain (unchangeable directly), aur inka size fixed hota hai. Examples:

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `bigint`
- `symbol`

✓ Reference Types (Sanketik ya Referenced Data):

Reference types wo hote hain jo by reference store hote hain — iska matlab hai variable us data ki memory location ko point karta hai.

Ye mutable hote hain (in-place change ho sakta hai).

Examples:

- `object`
- `array`
- `function`

✓ Primitive Type Example:

```
let a: number = 10;
let b = a;
b = 20;

console.log(a); // 10
console.log(b); // 20
```

`a` ki value `10` thi, jab humne `b = a` kiya, toh `b` ne ek copy li.

Baad mein jab `b = 20` kiya, toh `a` par koi effect nahi pada. Kyunki primitive by value hota hai.

✓ Reference Type Example:

```
let obj1 = { name: "Piyush" };
let obj2 = obj1;
obj2.name = "Gupta";

console.log(obj1.name); // Gupta
console.log(obj2.name); // Gupta
```

`obj2` ne `obj1` ka reference copy kiya, toh dono same memory address ko point kar rahe hain.

Isiliye jab `obj2.name` badla, `obj1.name` bhi change ho gaya.

3. Most Asked Interview Questions

? Q1: What is the difference between primitive and reference types in TypeScript?

Answer:

Primitive types store actual value (jaise `number`, `string`), while reference types store the reference (address) to the data in memory (like `object`, `array`).

Primitive types are immutable; reference types are mutable.

? Q2: Is `null` a primitive or reference type?

Answer:

`null` is a primitive type in TypeScript, even though it may look like a reference.

? Q3: How does TypeScript differentiate between primitive and reference types?

Answer:

At runtime, JavaScript (jisme TypeScript compile hota hai) internally decides storage —

- Primitive values are stored directly in memory,
 - Reference values are stored as a reference (pointer) to an object in heap memory.
-

Bonus Tip:

- TypeScript allows explicit typing, so you can define your primitives safely:

```
let userName: string = "Piyush";  
let isLoggedIn: boolean = true;  
let score: number = 99.9;
```

Array in TypeScript

TypeScript mein **array** ek aisa data structure hota hai jisme **multiple values of same type** ko ek jagah store kiya ja sakta hai.

TypeScript ka power ye hai ki hum array ke andar ke elements ka type bhi define kar sakte hain.

```
let arr: number[] = [1, 2, 3];
```

```
let arr: Array<number> = [1, 2, 3];
```

Dono syntax same kaam karte hain, bas likhne ka tareeka alag hai.

2. Types of Arrays

```
let fruits: string[] = ["apple", "banana", "mango"];
```

```
let mixed: (string | number)[] = ["Piyush", 22, "MERN"];
```

Example 1: String Array

```
let names: string[] = ["Piyush", "Gupta"];
names.push("Developer");
console.log(names); // ["Piyush", "Gupta", "Developer"]
```

Example 2: Array of Objects

```
type User = {
  name: string;
  age: number;
};

let users: User[] = [
  { name: "Piyush", age: 22 },
  { name: "Ravi", age: 24 },
];
```

Example 3: Array with Union Types

```
let data: (string | number)[] = ["A", 1, "B", 2];
```

Interview Questions

? Q1: How do you define an array in TypeScript?

Answer:

We can define arrays in two ways:

- `let arr: number[] = [1, 2, 3];`
- `let arr: Array<number> = [1, 2, 3];`

? Q2: Can you store multiple types in a TypeScript array?

Answer:

Yes, using union types:

```
let arr: (number | string)[] = [1, "hello"];
```

? Q3: How to define an array of custom objects?

Answer:

```
type Book = { title: string; price: number };

let books: Book[] = [

  { title: "TS Handbook", price: 100 },

  { title: "JS Bible", price: 200 },

];
```

! Bonus Tip:

- Agar aap array ke elements ke types ko **strict** rakhna chahte ho toh **as const** ka use karo.

```
const arr = [1, 2, 3] as const;
```

```
// ab arr readonly ban gaya
```

- TypeScript mein aap **readonly** arrays bhi bana sakte ho:

```
let readonlyArr: readonly number[] = [1, 2, 3];
```

```
// readonlyArr.push(4); ❌ error
```

🔍 as const exact values ko bhi fix kar deta hai ([10, 20, 30])

🔍 readonly bas type ko restrict karta hai, value koi bhi number ho sakta hai.

```
const data = [10, 20, 30] as const;
// type => readonly [10, 20, 30]

let data2: readonly number[] = [10, 20, 30];
// type => readonly number[]
```

Tuples in TypeScript

Tuples TypeScript mein ek **fixed-length array** hote hain jisme **har position ka specific type** define hota hai.

```
let person: [string, number] = ["Piyush", 22];
```

Real-World Example:

```
type User = [id: number, username: string, isAdmin: boolean];  
  
const user1: User = [101, "piyush", true];
```

Tuple with Optional & Rest:

```
let tuple: [string, number?, ...boolean[]];  
  
tuple = ["hello"];  
tuple = ["hi", 5, true, false, true];
```

Q: How is a tuple different from an array in TypeScript?

✅ **Answer:**

In tuple, each element has a fixed **type and position**, whereas in arrays all elements share the **same type**.

Enums in TypeScript

Enum (Enumeration) ek user-defined type hota hai jisme **named constants** store kiye jaate hain.

Yeh mostly tab use hota hai jab kisi variable ka **limited set of values** ho (jaise `status`, `roles`, `direction`).

```
enum BookingStatus {  
  NEW = "New",  
  ASSIGNED = "Assigned",  
  COMPLETED = "Completed",  
  CANCELLED = "Cancelled"  
}  
  
let current: BookingStatus = BookingStatus.NEW;
```

Interview Questions:

? Q1: What is an Enum in TypeScript and when should you use it?

✓ Answer: Enum is a way to define a set of named constants. It is used when a variable should only hold a specific set of values like `"PENDING"`, `"SUCCESS"`, `"FAILED"` etc.

? Q2: What is the difference between numeric and string enums?

✓ Answer:

- Numeric enums assign **incremental numbers by default**, starting from 0.
- String enums assign **explicit string values**, which are better for debugging/logging.

TypeScript ke *sabse confusing aur interview-favorite types*

1. any

Definition:

any type ka matlab hai "TypeScript bhool jao, kuch bhi chalega."
TypeScript ki type-checking off ho jaati hai.

```
let value: any = 10;  
value = "hello";  
value = true;
```


Avoid in real projects unless 100% needed.

2. unknown

Definition:

unknown = safer version of **any**.

TypeScript ko pata hai value ka type unknown hai, but use karne se pehle **type-check karna padta hai**.

```
let value: unknown = "piyush";  
  
if (typeof value === "string") {  
  console.log(value.toUpperCase()); //  safe  
}
```

Q: **any** vs **unknown**?

- **any**: kuch bhi kar sakte ho, bina check ke
- **unknown**: use karne ke pehle **type check compulsory**

3. void

Definition:

`void` ka use hota hai functions ke liye jo **kuch return nahi karte**.

```
function greet(): void {  
  console.log("Hello Piyush");  
}
```

4. null & undefined

`null`: matlab value intentionally absent hai

```
let user: string | null = null;
```

`undefined`: variable defined hai but value **nahi mili**

```
let age: number | undefined;
```

5. never

Definition:

`never` ka matlab hai "yeh value kabhi exist nahi karegi"

Use hota hai:

- infinite loops
- errors
- unreachable code

```
function crash(): never {  
    throw new Error("Something went wrong");  
}
```

Interview Rapid Fire:

? Q: When should you use **unknown** instead of **any**?

✓ Jab aapko pata hai value dynamic hai, par aap safety maintain karna chahte ho.

? Q: What is the return type of a function that throws an error?

✓ **never**

? Q: Difference between **undefined** and **null**?

✓ **undefined** ka matlab "value abhi tak assign nahi hui",
null ka matlab "value intentionally empty hai".

Interfaces in TypeScript

1. Definition:

Interface ek TypeScript ka feature hai jo ek object ke **structure** ko define karta hai — matlab kya keys honge, unka type kya hoga, aur kya mandatory hai.

Simple Shabdon Mein:

➡ Interface = “Contract” ya “Blueprint” jo bataata hai ki object kaisa dikhna chahiye.

```
interface User {  
  name: string;  
  age: number;  
  isAdmin: boolean;  
}  
  
const user1: User = {  
  name: "Piyush",  
  age: 22,  
  isAdmin: true,  
};
```

Interface with Inheritance (extends):

```
interface Person {  
  name: string;  
}  
  
interface Employee extends Person {  
  employeeId: number;  
}  
  
const emp: Employee = {  
  name: "Piyush",  
  employeeId: 101,  
};
```

5. Interview Questions (Must-know)

? Q1: What is the difference between **interface** and **type**?

✓ Interface sirf object structure define karta hai, jabki **type union, intersection, primitive, object** sab define kar sakta hai.

Interface is **extendable**, type mein bhi ye ho sakta hai but thoda limited.

? Q2: Can an interface extend another interface?

✓ Yes, using **extends**.

? Q3: Can we use interfaces with classes?

✓ Haan, interface se class ko bind karke ensure kar sakte ho ki us class mein required properties ya methods hon.

1. Type Aliases (**type**)

Definition:

type keyword se hum **custom naam se type define** karte hain — jise baar-baar reuse kiya ja sakta hai.

Yeh **primitive, union, function, array, object**, ya **intersection** sab kuch hold kar sakta hai.

👉 Primitive Type Alias:

```
type UserId = number;  
let id: UserId = 101;
```

👉 Object Type Alias:

```
type User = {  
  name: string;  
  age: number;  
};
```

👉 Function Type Alias:

```
type Greet = (name: string) => string;  
  
const greetUser: Greet = (name) => `Hello, ${name}`;
```

2. Intersection Types (&)

■ Definition:

Intersection type ka use hota hai jab aapko **do ya zyada types ko merge** karna ho — matlab ek object ya variable dono types ke properties fulfill kare.

```
type Person = {  
  name: string;  
};  
  
type Employee = {  
  employeeId: number;  
};  
  
type FullEmployee = Person & Employee;  
  
const emp: FullEmployee = {  
  name: "Piyush",  
  employeeId: 101,  
};
```

Interview Questions

? **Q1: What's the difference between `type` and `interface`?**

✓ `type` is more flexible (can define union, intersection, function types),
`interface` is mainly for object structure and can merge with same name.

? **Q2: What is an intersection type used for?**

✓ When you want to **combine multiple types** and the final object should satisfy **all of them**.

? **Q3: Can `type` extend another `type` or `interface`?**

✓ Yes, using intersections (&)

Topic 1: Class Definition in TypeScript

Definition:

Class ek blueprint hoti hai jiske base pe hum multiple objects create kar sakte hain — har object same properties aur methods share karta hai.

TypeScript mein `class` ka use hota hai real-world objects ko represent karne ke liye — jaise User, Product, Vehicle, etc.

```
class Product {  
  name: string;  
  price: number;  
  
  printDetails() {  
    console.log(`Product: ${this.name}, Price: ₹${this.price}`);  
  }  
}  
  
const item = new Product();  
item.name = "Laptop";  
item.price = 50000;  
item.printDetails(); // Product: Laptop, Price: ₹50000
```

TypeScript mein class ke andar likhi properties ko **declare karna mandatory hai** (agar aap `strict` mode on rakhte ho)

Topic 2: Constructors in TypeScript

Definition:

Constructor ek **special method** hota hai class ke andar jo object banne ke time **automatically call** hota hai — aur mostly properties ko initialize karta hai.

Class me sabse pehle constructor run karta hai

```
class Person {
  name: string;
  age: number;

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


  greet() {
    console.log(`Hello, I am ${this.name} and I'm ${this.age}`);
  }
}

const p1 = new Person("Piyush", 22);
p1.greet(); // Hello, I am Piyush and I'm 22
```

(Short-hand):

```
class User {
  constructor(public name: string, private age: number) {}
}
```

 Q: What is a constructor in TypeScript?

 It's a special function in a class that gets called when an object is created, usually used to initialize properties.

Feature	TypeScript	JavaScript
Constructor parameter types	✓ Required (e.g., <code>name: string</code>)	✗ Not available
Compile-time safety	✓ Type checked	✗ No type checking
Access modifiers in constructor	✓ (e.g., <code>private</code> , <code>public</code>)	✗ Not supported
Parameter properties	✓ Supported (<code>public name: string</code>)	✗ Not supported

Topic 3: Access Modifiers in TypeScript

(public, private, protected)

Definition:

Access modifiers define karte hain ki class ke members (properties ya methods) ko **kaun access kar sakta hai**.

JavaScript mein ye **available nahi** hote (except `#private` since ES2022), but TypeScript mein ye **by default supported** hain.

Modifier	Accessible From Where?
<code>public</code>	Everywhere (default if not specified)
<code>private</code>	Only inside the same class
<code>protected</code>	Same class + subclasses (child classes)

```

class Person {
  public name: string;
  private ssn: string;
  protected age: number;

  constructor(name: string, ssn: string, age: number) {
    this.name = name;
    this.ssn = ssn;
    this.age = age;
  }

  public showName() {
    return this.name;
  }

  private showSSN() {
    return this.ssn;
  }

  protected showAge() {
    return this.age;
  }
}

```

```

class Student extends Person {
  getAge() {
    return this.age; // ✅ allowed because protected
    // return this.ssn; ❌ Error: private
  }
}

```

? **Q1: What is the difference between `private` and `protected` in TypeScript?**

✅ `private` members are accessible **only within the class**, while `protected` members are accessible **within the class and its subclasses**.

? **Q2: What is the default access modifier in TypeScript?**

✅ `public` is the default modifier.

Topic 4: Readonly Properties in TypeScript

Definition:

readonly keyword ka use kisi property ko **immutable** (non-changeable) banane ke liye hota hai **after initialization**.

Ek baar value set ho gayi, uske baad usse modify nahi kar sakte.

```
class User {  
  readonly id: number;  
  name: string;  
  
  constructor(id: number, name: string) {  
    this.id = id;      // ✅ allowed inside constructor  
    this.name = name;  
  }  
  
  updateId(newId: number) {  
    // this.id = newId; ❌ Error: Cannot assign to 'id'  
  }  
}
```

Key Rule:

- **readonly** property ko **sirf constructor ke andar** assign kiya ja sakta hai.
- Baad mein (class ke methods ya bahar) **change nahi kiya ja sakta**.

JS mein readonly banane ke liye `Object.freeze()` ya `defineProperty()` use karna padta hai — which is **messy**.


Topic 5: Optional Properties in TypeScript

Definition:

TypeScript mein kisi property ko **optional** banane ke liye uske naam ke baad **?** lagate hain.

Iska matlab hai — wo property **ho bhi sakti hai, nahi bhi**.

```
class User {  
  name: string;  
  age?: number; // optional property  
  
  constructor(name: string, age?: number) {  
    this.name = name;  
    this.age = age;  
  }  
}
```

```
const u1 = new User("Piyush", 22);  
const u2 = new User("Gupta"); //  allowed (age not passed)
```

Optional properties ka type actually **type | undefined** hota hai.

? Q: What is the type of an optional property like **age?: number**?

 It becomes **number | undefined**

Topic 6: Parameter Properties in TypeScript

Definition:

Parameter Properties ek TypeScript ki **shorthand feature** hai jisse hum constructor ke andar hi:

- property declare kar sakte ho
 - access modifier laga sakte ho
 - aur usse initialize bhi kar sakte ho
- Ek hi line mein sab kuch!**

```
class User {  
  constructor(public name: string, public age: number) {}  
}
```

```
class Account {  
  constructor(  
    private readonly accountNo: string,  
    public balance: number  
  ) {}  
}
```

Topic 7: Inheritance in TypeScript

Inheritance ek concept hai jisme ek class (child class) doosri class (parent class) ke properties aur methods ko **reuse kar sakti hai** using **extends** keyword.

Yeh concept DRY (Don't Repeat Yourself) principle follow karta hai.

```
class Animal {
  name: string;

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

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  breed: string;

  constructor(name: string, breed: string) {
    super(name); // 🐾 calls parent constructor
    this.breed = breed;
  }

  bark() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog("Tommy", "Labrador");
dog.speak(); // Tommy makes a noise.
dog.bark();  // Tommy barks.
```

extends

Child class inherits from parent

super()

Call to parent class constructor

Topic 8: Getter and Setter in TypeScript

Definition:

Getters (get) aur **Setters (set)** ka use hum class properties ke **access** aur **update** karne ke liye karte hain — lekin with **extra control ya logic**.

Isse hum:

- Private property ko safely access kar sakte hain
- Update karne se pehle validation daal sakte hain

```
class Person {
  private _name: string = "";

  // getter
  get name(): string {
    return this._name;
  }

  // setter
  set name(newName: string) {
    if (newName.length < 2) {
      throw new Error("Name too short");
    }
    this._name = newName;
  }
}

const p = new Person();
p.name = "Piyush"; // ✅ setter called
console.log(p.name); // ✅ getter called
```


Topic 9: Static Properties and Methods in TypeScript

Definition:

`static` keyword ka use kisi **class ke aise member** (property ya method) ko define karne ke liye hota hai jise **class ke naam se hi access kiya jaa sake** — bina object banaye.

Yeh members class ke sabhi objects ke **common** hote hain (shared across instances).

```
class MathUtils {  
  static pi: number = 3.14159;  
  
  static square(x: number): number {  
    return x * x;  
  }  
}
```

```
console.log(MathUtils.pi);           // ✓ 3.14159  
console.log(MathUtils.square(5));    // ✓ 25
```

👉 Yahan `pi` aur `square()` dono ko humne class ke naam se directly access kiya — bina kisi `new` object ke.

Access via object = Error:

```
const m = new MathUtils();
```

```
// m.square(2);  Error: Property 'square' is a static member
```

? Q: Can static methods access non-static properties?

✗ No. Because non-static members belong to instances, not the class.

What are Function Types in TypeScript?

In TypeScript, **Function Types** let you define:

1. **What parameters a function must take**
2. **What it should return**

 Example 1: Basic Function with Types

```
function multiply(a: number, b: number): number {  
    return a * b;  
}
```

Explanation:

- `a: number` and `b: number` → the function accepts two numbers.
- `: number` at the end → the function returns a number.

 Example 2: Function Type Assigned to a Variable

```
let greeting: (name: string) => string;  
  
greeting = function(name: string) {  
    return "Hello " + name;  
};
```

Explanation:

- The variable `greeting` is declared as a function that:
 - takes a single `string` argument
 - returns a `string`

✓ Optional Parameters

- Use `?` after the parameter name to make it optional.
- If the optional parameter is not provided, its value will be `undefined`.

✓ Default Parameters

- You can assign a **default value** to a parameter.
- If the caller does not pass a value, the default is used.

```
7 function abcdef(name: string, age: number, gender: string = "not to be disclosed") {
8     console.log(name, age, gender);
9 }
10
11 abcdef("harsh", 25, "male");
12 abcdef("lagbataq", 22);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
→ typescript-domination node app.js
harsh 25 male
lagbataq 22 not to be disclosed
→ typescript-domination
```

Rest Parameters – TypeScript mein

Jab humein nahi pata ki function ke andar **kitne arguments aayenge**, tab hum rest parameter use karte hain.

Ye sari values ko ek array ke form mein **collect** kar leta hai.

```
function sum(...numbers: number[]) {  
    return numbers.reduce((total, num) => total + num, 0);  
}
```

```
sum(1, 2);           // Output: 3  
sum(10, 20, 30, 40); // Output: 100  
sum();               // Output: 0
```

Toh `...numbers` ka matlab hai jitne bhi numbers pass karoge, wo sab ek array ban jaayenge. Fir hum `reduce()` se sabka sum kar rahe hain.

Important Baaten:

- `...` (triple dots) ka use karte hain rest parameter ke liye.
- **Hamesha last parameter** hi rest ho sakta hai. Beech mein allowed nahi hai.
- Sirf ek hi rest parameter ho sakta hai ek function mein.
- Uska type array hona chahiye – jaise `number[]`, `string[]`, etc.

```
function greet(message: string, ...names: string[]) {  
    for (let name of names) {  
        console.log(`${message}, ${name}`);  
    }  
}  
  
greet("Hello", "Piyush", "Amit", "Riya");
```

Hello, Piyush

Hello, Amit

Hello, Riya

Function Overloading in TypeScript

Function Overloading ka matlab hai **ek hi function ka naam** use karke **alag-alag parameters** ke sath alag behavior define karna.

TypeScript mein hum define kar sakte hain ki agar:

- kuch specific parameters aayein → to kya output hoga,
- aur agar doosre aayein → to kya output hoga.

Yaani ek function ka **multiple signature** banate hain. Jaise ek actor ka alag-alag role hota hai depending on scene 😊

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

- Pehle 2 lines – **function overload signatures** hain.
- Teesri line – **actual implementation** hai.
- TypeScript decide karega kaunsa overload chalana hai based on arguments.

```
console.log(add(5, 10));           // Output: 15
console.log(add("Hello ", "Piyush")); // Output: Hello Piyush
```

Points to Yaad Rakho:

- Overload sirf **function declaration** pe hota hai, not on arrow functions.
- Sirf **first match** wala overload chalta hai. So ordering bhi important hoti hai.

```

7 // ts fnc signature
8 function abcd(a: string): void;
9 function abcd(a: string, b: number): number;
10
11 function abcd(a: any, b?: any) {
12     if (typeof a === "string" && b === undefined) {
13         console.log("hey");
14     }
15     if (typeof a === "string" && typeof b === "number") {
16         return 123;
17     }
18     else throw new Error("something is wrong");
19 }
20
21 abcd("hey");
22 abcd("hey", 12);

```

Generic Functions in TypeScript

◆ Definition:

Generic Function wo function hota hai jo data type ko dynamic banata hai, taki ek hi function multiple data types ke saath kaam kar sake without rewriting it for each type.

```

function functionName<T>(parameter: T): T {
    return parameter;
}

```

T is a placeholder for a **type** (generic type parameter).

T ka naam kuch bhi ho sakta hai (T, U, K, V, etc), but T is conventionally used.

✓ Example: Without Generics (bad way)

```
function logger(a: string | null | undefined | string | number) {  
  
}  
  
logger("hey");  
logger(12);  
logger(undefined);  
logger(null);
```

👉 Har type ke liye alag function likhna pad raha hai — ✗ bad practice.

```
function log<T>(val: T) {  
    console.log(val);  
}  
  
log(12);
```

Generic Function with Arrays

```
function getFirstElement<T>(arr: T[]): T {  
    return arr[0];  
}  
  
const firstNum = getFirstElement([1, 2, 3]); // number  
const firstStr = getFirstElement(["a", "b", "c"]); // string
```

Generic Interface in TypeScript

Definition (Samjho Pehle):

Generic Interface ek aisa interface hota hai jisme tum type ko dynamic (yaani ki generic) bana sakte ho. Iska matlab, interface banate waqt tum ye fix nahi karte ki kaunsa data type hoga — use later decide kiya ja sakta hai jab tum usse implement karoge.

Why Use Generic Interface?

- **Reusability:** Har baar alag type ke liye alag interface nahi banana padta.
- **Type Safety:** TypeScript khud check karta hai ki tumne correct type diya hai ya nahi.
- **Flexibility:** Tum kisi bhi type ke data ke sath kaam kar sakte ho.

```
interface Box<T> {  
  value: T;  
  show: () => void;  
}
```

Yahaan **T** ek placeholder hai — tum jab **Box** ko use karoge tab **T** ki jagah koi bhi type de sakte ho jaise **number**, **string**, **boolean**, custom object, etc.

```
interface Halua<T> {  
  name: string;  
  age: number;  
  key: T;  
}  
  
function abcd(obj: Halua<string>) {  
  obj.key = obj.key  
}  
  
abcd({ name: "foo", age: 25, key: "ahjscasbc" })
```


Generic Classes in TypeScript

Definition:

Generic Class ek aisi class hoti hai jo **type-safe** aur **reusable** banayi ja sakti hai by using generics.

Matlab class banate waqt hum type fix nahi karte, balki use karte waqt batate hain ki kis type ka data usme hoga.

```
class BottleMaker<T> {  
    constructor(public key: T) { }  
}  
  
let b1 = new BottleMaker<string>("hey");  
console.log(b1);
```

T ek **generic type parameter** hai.

Jab aap `BottleMaker` class ka object banate ho, to aap decide karte ho ki **T** kya hoga (e.g., `string`, `number`, `boolean`, etc.).

Constructor me `key` usi type ka hoga jo aap **T** me doge.

```
let b1 = new BottleMaker<string>("hey");  
console.log(b1);
```

Modules in TypeScript?

```
export default class Payment {  
  constructor(public price: number) { }  
}
```

```
import Payment from "../payment";  
  
let a = new Payment(12);  
let b = new Payment(122);  
console.log(a, b);
```

```
async@Bumblebee typescript-domination % node app.js  
Payment { price: 12 } Payment { price: 122 }  
async@Bumblebee typescript-domination %
```

Type Assertion (As Operator)

Jab tum TypeScript ko **force** karte ho ki "bhai tu maano ki ye variable is type ka hi hai", tab **type assertion** hota hai.

```
let value: any = "Hello World";  
let length: number = (value as string).length;
```

👉 Yahan pe hum TypeScript ko bata rahe hain ki `value` actually `string` hai. Uske baad hi `.length` property safely lagayi.

Kab use karte hain?

- Jab koi value **any** ya **unknown** type ki ho.
- Jab API se response aaye aur tum sure ho ki woh specific structure ka hi hai.

✗ Galtiyan jo log karte hain:

`let num = "123" as number; // ✗ Error: Cannot convert 'string' to 'number' directly`

Yahan pe TypeScript forcefully string ko number nahi banata. Ye sirf *compiler* ke liye type assume karta hai, actual conversion nahi karta!

Use type conversion for such cases:

```
let num = Number("123");
```