⍰ TypeScript Full Course Roadmap

We'll learn in 4 phases:

✅PHASE 1: Basics & Fundamentals

1. [⍰] What is TypeScript & Setup (you already did this ✅)
2. **Basic Types**: `string`, `number`, `boolean`, `any`, `unknown`
3. **Functions**: with parameter & return types
4. **Arrays & Tuples**
5. **Enums**
6. **Type Aliases & Union Types**
7. **Literal Types & Type Inference**

⍰ PHASE 2: Intermediate Concepts

1. **Interfaces**
2. **Optional Properties & Readonly**
3. **Type vs Interface**
4. **Narrowing & Type Guards**
5. **Type Assertions**

⍰ PHASE 3: Advanced Concepts

1. **Classes & Inheritance**
2. **Access Modifiers**: `public`, `private`, `protected`
3. **Generics**
4. **Decorators (optional)**
5. **Utility Types (`Partial`, `Pick`, etc.)**

⍰ PHASE 4: Projects & Real-World Usage

1. **Using TypeScript with Node.js**
2. **Using TypeScript with React (optional)**
3. **Build a small project**
4. **How to use `.d.ts` (type definition) files**

⍰ What Is TypeScript?

TypeScript is a **superset of JavaScript** — which means it's just like JavaScript, but with **optional static typing**.

⍰ Why Use TypeScript?

- Catches errors at **compile-time** instead of runtime.
- Makes code easier to understand with **type annotations**.
- Great for big apps or working in teams.

⍰ Step 1: Install TypeScript

✓Prerequisites

Make sure you have **Node.js** installed. You can check using:

```
bashCopyEditnode -v
npm -v
```

⍰ Install TypeScript globally:

Open your terminal and run:

```
bash


CopyEdit
npm install -g typescript
```

This gives you access to the `tsc` command (TypeScript compiler).

⍰ Phase 1 - Step 2: Basic Types

Let's begin with **Basic Types**!

⍰ What are Types?

TypeScript gives variables a **type**, so errors can be caught **before running** the code.

✓Common Basic Types

Type	Example
`string`	`"hello"`
`number`	`5`, `3.14`
`boolean`	`true`, `false`
`any`	Anything (avoid if possible)
`unknown`	Like any but safer
`null` / `undefined`	null values

⍰ Example Code

```
tsCopyEdit// string
let name: string = "Ankit";

// number
let age: number = 21;

// boolean
let isStudent: boolean = true;

// any (not recommended often)
let randomValue: any = "Hello";
randomValue = 100;

// unknown (better than any)
let input: unknown;
input = "Something";
input = 123;

// null & undefined
let a: null = null;
let b: undefined = undefined;
```

⍰ Try it Yourself (Practice Time)

In your `index.ts`, write this:

```
tsCopyEditlet firstName: string = "YourName";
let rollNo: number = 123;
let isPassed: boolean = true;

console.log("Student:", firstName, "Roll No:", rollNo, "Passed?", isPassed);
```

Then compile & run:

```
bashCopyEdittsc index.ts
node index.js
```

✓Task

1. Create variables with these types:
   - a `string` for your favorite color
   - a `number` for your birth year
   - a `boolean` for “do you like coding?”
2. Log them to the console.

✅Step 3: Functions with Types

In TypeScript, you can add types to:

- Function **parameters**
- Function **return value**

This helps prevent bugs and makes your code clearer.

⍰ Syntax

```
tsCopyEditfunction functionName(parameter: type): returnType {
  // logic
}
```

⍰ Example 1: Function with typed parameters

```
tsCopyEditfunction greet(name: string): void {
  console.log("Hello, " + name);
}

greet("Ankit");
```

void means this function doesn’t return anything.

⍰ Example 2: Function with return type

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

let result = add(10, 20);
console.log("Sum is:", result);
```

⍰ Example 3: Function with default parameter

```
tsCopyEditfunction welcome(name: string = "Guest"): string {
  return "Welcome, " + name;
}

console.log(welcome("Ankit"));
console.log(welcome());
```

⍰ Example 4: Arrow Function

```
tsCopyEditconst multiply = (x: number, y: number): number => {
  return x * y;
};

console.log(multiply(3, 4));
```

⍰ Practice Time

In your `index.ts`, try writing these functions:

1. A function square that takes a number and returns its square.
2. A function `isEven` that returns `true` if a number is even, otherwise `false`.

Example format:

```
tsCopyEditfunction square(n: number): number {
  return n * n;
}
```

✅Task

Write and test the following:

```
tsCopyEdit// square(5) should return 25
// isEven(4) should return true
```

Compile & run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 4: Arrays & Tuples

⍰ 1. Arrays in TypeScript

You can specify the type of elements in an array using either:

```
tsCopyEditlet arr: number[] = [1, 2, 3];
let names: string[] = ["Ankit", "Raj"];
```

Or using **generic syntax**:

```
ts
```

```
CopyEdit
let values: Array<number> = [10, 20, 30];
```

▯ Example: Array Usage

```
tsCopyEditlet fruits: string[] = ["apple", "banana", "mango"];
console.log(fruits[0]); // apple

fruits.push("orange");
console.log(fruits); // ["apple", "banana", "mango", "orange"]
```

▯ 2. Tuples in TypeScript

A **tuple** is an array with **fixed length and specific types** in a defined order.

```
ts


CopyEdit
let person: [string, number] = ["Ankit", 21];
```

You must follow the order and types strictly:

```
tsCopyEdit// Correct
let employee: [number, string] = [101, "John"];

// ❌Incorrect
// let wrong: [string, number] = [25, "Ankit"]; // Error
```

▯ Example: Tuple Usage

```
tsCopyEditlet user: [string, boolean] = ["admin", true];
console.log(user[0]); // "admin"
console.log(user[1]); // true
```

▯ Practice Time

Try the following in your `index.ts`:

1. Create a `number[]` array of your favorite 3 numbers.
2. Create a `string[]` array of 3 city names.
3. Create a tuple with your `[name: string, age: number, isDeveloper: boolean]`.
4. Log all values.

✓Task Example:

```
tsCopyEditlet luckyNumbers: number[] = [7, 11, 21];
let cities: string[] = ["Delhi", "Mumbai", "Jaipur"];

let myProfile: [string, number, boolean] = ["Ankit", 21, true];

console.log("Numbers:", luckyNumbers);
console.log("Cities:", cities);
console.log("Profile:", myProfile);
```

Compile and run:

```
bashCopyEdittsc index.ts
node index.js
```

✓Step 5: Enums (Enumerations)

⍰ What is an Enum?

An **enum** is a way to give **friendly names** to sets of **numeric or string values**.

Useful when you have a fixed set of values (like days of the week, user roles, etc.)

⍰ Syntax

```
tsCopyEditenum Role {
  User,
  Admin,
  SuperAdmin,
}
```

> By default, `User = 0`, `Admin = 1`, and so on.

⍰ Example 1: Numeric Enum

```
tsCopyEditenum Direction {
  North, // 0
  South, // 1
  East,  // 2
  West   // 3
}

let dir: Direction = Direction.South;
console.log(dir); // Output: 1
```

⍰ Example 2: Custom Values

```
tsCopyEditenum Status {
  Success = 200,
  NotFound = 404,
  ServerError = 500
}

let code: Status = Status.NotFound;
console.log(code); // 404
```

⍰ Example 3: String Enum

```
tsCopyEditenum Theme {
  Light = "LIGHT",
  Dark = "DARK",
  System = "SYSTEM"
}

let currentTheme: Theme = Theme.Dark;
console.log(currentTheme); // DARK
```

⍰ Practice Time

In your `index.ts` file:

1. Create an enum for the **days of the week** (Monday to Sunday).
2. Print the value of **Wednesday**.
3. Create another string enum for **user roles**: `Admin`, `Editor`, `Viewer`.

✅Task Example

```
tsCopyEditenum Days {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
}

console.log("Midweek:", Days.Wednesday); // Output: 2

enum UserRole {
  Admin = "ADMIN",
  Editor = "EDITOR",
  Viewer = "VIEWER"
}

let user: UserRole = UserRole.Editor;
console.log("User Role:", user); // Output: EDITOR
```

Then compile and run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 6: Type Aliases & Union Types

⍰ 1. Type Aliases

A **type alias** lets you create a **custom name** for a type. This is helpful when a type is used many times or is complex.

⍰ Syntax

```
tsCopyEdittype Name = string;

let userName: Name = "Ankit";
```

⍰ Example: Custom Object Type

```
tsCopyEdittype User = {
  name: string;
  age: number;
  isAdmin: boolean;
};

let user1: User = {
  name: "Raj",
  age: 25,
  isAdmin: false,
};

console.log(user1);
```

⍰ 2. Union Types

With **union types**, a variable can be **more than one type**.

⍰ Syntax

```
tsCopyEditlet id: string | number;

id = 101;       // ✅allowed
id = "101abc";  // ✅allowed
```

⍰ Example: Function Using Union

```
tsCopyEditfunction printId(id: string | number): void {
  console.log("Your ID is:", id);
}

printId(101);
printId("abc123");
```

⍰ 3. Combining Both

```
tsCopyEdittype Score = number | "pass" | "fail";

let result: Score = 95;
result = "pass"; // ✅
```

⍰ Practice Time

1. Create a type alias `Student` with `name: string`, `roll: number`, `isPassed: boolean`.
2. Create a variable using that type.
3. Create a function `showId` that accepts `string | number` and logs it.
4. Create a type `Result = "pass" | "fail" | number`, and use it.

✅Task Example

```
tsCopyEdittype Student = {
  name: string;
  roll: number;
  isPassed: boolean;
};

let s1: Student = {
  name: "Ankit",
  roll: 23,
  isPassed: true
};

console.log(s1);

function showId(id: string | number): void {
  console.log("ID is:", id);
}

showId(404);
showId("XYZ123");

type Result = "pass" | "fail" | number;
let examResult: Result = "pass";
examResult = 85;

console.log("Result:", examResult);
```

Compile and run:

```
bashCopyEdittsc index.ts
node index.js
```

✓Step 7: Literal Types & Type Inference

⍰ 1. Literal Types

Literal types allow you to specify the **exact value** a variable can have.

⍰ Syntax

```
tsCopyEditlet direction: "left" | "right" | "center";

direction = "left";    // ✓
direction = "up";      // ✗Error
```

⍰ Example

```
tsCopyEdittype ButtonSize = "small" | "medium" | "large";

let size: ButtonSize = "medium";
console.log("Button Size:", size);
```

> This is great for dropdowns, roles, modes, etc.

⍰ 2. Type Inference

TypeScript can **automatically guess** the type based on the value you assign.

⍰ Example

```
tsCopyEditlet score = 100;      // Inferred as number
let name = "Ankit";   // Inferred as string
let passed = true;    // Inferred as boolean

// ✕name = 123; // Error because it's inferred as string
```

⍰ 3. const + Literal Types

When you use const, TypeScript infers the **literal value** as the type.

```
ts


CopyEdit
const role = "admin"; // type is "admin" not just string
```

⍰ Example: Literal in Function

```
tsCopyEditfunction setTheme(mode: "light" | "dark") {
  console.log("Theme set to:", mode);
}

setTheme("light"); // ✓
setTheme("auto");  // ✕
```

⍰ Practice Time

1. Create a literal type `Theme = "light" | "dark" | "system"`.
2. Write a function `applyTheme(theme: Theme)` that prints the theme.
3. Use `const` to create a variable that holds the value `"system"` and pass it to the function.

✓Task Example

```
tsCopyEdittype Theme = "light" | "dark" | "system";

function applyTheme(theme: Theme): void {
  console.log("Applied theme:", theme);
}

const myTheme = "system"; // inferred as "system"
applyTheme(myTheme); // ✓
```

Compile and run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 8: Interfaces vs Type Aliases

⍰ 1. What is an Interface?

An **interface** defines the **structure** of an object, like a blueprint.

```
tsCopyEditinterface Person {
  name: string;
  age: number;
  isStudent: boolean;
}
```

You can use it like this:

```
tsCopyEditconst p1: Person = {
  name: "Ankit",
  age: 21,
  isStudent: true,
};
```

⍰ 2. Type Alias for Objects

You can do the same with `type`:

```
tsCopyEdittype PersonType = {
  name: string;
  age: number;
  isStudent: boolean;
};
```

Both `interface` and `type` are similar for object definitions.

⍰ Difference Between `interface` and `type`

Feature	`interface`	`type`
Extending other types	✅(can extend multiple interfaces)	✅(can use intersection &)
Can define primitives/unions	❌	✅
Declaration merging	✅(can merge multiple declarations)	❌(type can’t be re-declared)

⍰ Example: Extending Interface

```
tsCopyEditinterface Animal {
  name: string;
}

interface Dog extends Animal {
  breed: string;
}

const dog1: Dog = {
  name: "Tommy",
  breed: "Labrador",
};
```

⍰ Example: Type Intersection

```
tsCopyEdittype Animal = {
  name: string;
};

type Dog = Animal & {
  breed: string;
};

const dog2: Dog = {
  name: "Rocky",
  breed: "German Shepherd",
};
```

⍰ Practice Time

1. Create an `interface` called `Car` with `brand`, `year`, and `electric: boolean`.
2. Create a `type` called `Bike` with `brand`, `cc`, and `gear: boolean`.
3. Print one object of each.

✓Task Example

```
tsCopyEditinterface Car {
  brand: string;
  year: number;
  electric: boolean;
}

const myCar: Car = {
  brand: "Tesla",
  year: 2023,
  electric: true,
};

type Bike = {
  brand: string;
  cc: number;
  gear: boolean;
};

const myBike: Bike = {
  brand: "Yamaha",
  cc: 150,
  gear: true,
};

console.log("Car:", myCar);
console.log("Bike:", myBike);
```

Compile and run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 9: Functions in TypeScript

□ 1. Basic Function with Types

You can add **types to parameters and return values** in a function.

□ Syntax

```
tsCopyEditfunction greet(name: string): string {
  return `Hello, ${name}`;
}
tsCopyEditconsole.log(greet("Ankit")); // ✅OK
// greet(123); ❌Error
```

□ 2. Function with Multiple Parameters

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

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

⍰ 3. Void Return Type

Use void when the function **does not return anything**.

```
tsCopyEditfunction logMessage(msg: string): void {
  console.log("Message:", msg);
}
```

⍰ 4. Optional Parameters

Add ? to make a parameter optional.

```
tsCopyEditfunction welcome(name: string, age?: number): void {
  console.log("Welcome", name);
  if (age) console.log("You are", age, "years old");
}
```

⍰ 5. Default Parameters

You can also set a **default value**:

```
tsCopyEditfunction multiply(a: number, b: number = 2): number {
  return a * b;
}

console.log(multiply(5)); // Output: 10
```

⍰ 6. Function Type Alias

You can even define the type of a function:

```
tsCopyEdittype AddFn = (a: number, b: number) => number;

const sum: AddFn = (x, y) => x + y;

console.log(sum(3, 4));
```

⍰ Practice Time

1. Create a function `subtract(a: number, b: number): number`.
2. Create a function `greetUser(name: string, city?: string): void`.
3. Create a function type alias `DivideFn = (a: number, b: number) => number` and use it.

✅Task Example

```
tsCopyEditfunction subtract(a: number, b: number): number {
  return a - b;
}

function greetUser(name: string, city?: string): void {
  console.log("Hello", name);
  if (city) {
    console.log("From", city);
  }
}

type DivideFn = (a: number, b: number) => number;

const divide: DivideFn = (a, b) => a / b;

console.log("Subtract:", subtract(10, 5));
greetUser("Ankit", "Delhi");
console.log("Divide:", divide(10, 2));
```

Compile and run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 10: Arrays, Tuples & `readonly`

□ 1. Arrays in TypeScript

You can specify the **type of items** inside the array.

```
tsCopyEditlet numbers: number[] = [1, 2, 3];
let names: string[] = ["Ankit", "John", "Aman"];
tsCopyEditnumbers.push(4);    // ✅
names.push("Sara"); // ✅
// names.push(100); // ❌Error: only strings allowed
```

□ 2. Readonly Arrays

Use `readonly` to prevent modifications:

```
tsCopyEditconst readonlyNames: readonly string[] = ["A", "B", "C"];

// readonlyNames.push("D"); // ❌Error
```

⍰ 3. Tuples (Fixed-length arrays with different types)

Tuples are arrays with **fixed length and types**.

```
tsCopyEditlet user: [string, number];

user = ["Ankit", 21]; // ✅
// user = [21, "Ankit"]; // ❌Wrong order
```

⍰ Example: Using Tuples

```
tsCopyEditlet product: [string, number, boolean] = ["Phone", 15000, true];

console.log("Product:", product[0]);  // "Phone"
console.log("Price:", product[1]);   // 15000
console.log("Available:", product[2]); // true
```

⍰ 4. Array of Tuples

```
tsCopyEditlet students: [string, number][];
students = [
  ["Ankit", 90],
  ["Ravi", 85],
  ["Priya", 95],
];
```

⍰ Practice Time

1. Create a number array called `marks`.
2. Create a tuple for a Book as `[title: string, pages: number]`.
3. Create a readonly array of colors.

✅Task Example

```
tsCopyEditlet marks: number[] = [80, 90, 85];
let book: [string, number] = ["Atomic Habits", 280];
const colors: readonly string[] = ["red", "green", "blue"];

console.log("Marks:", marks);
console.log("Book:", book);
console.log("Colors:", colors);
```

Compile and run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 11: Enums in TypeScript

⍰ 1. What is an Enum?

An **enum** lets you define a group of related constants with friendly names.

⍰ Basic Enum Syntax

```
tsCopyEditenum Direction {
  Up,
  Down,
  Left,
  Right,
}
```

- By default, `Up = 0`, `Down = 1`, etc.
- You can access values like: `Direction.Up` or get the name by value.

⍰ 2. Using Enums

```
tsCopyEditlet move: Direction = Direction.Left;
console.log(move); // Output: 2
```

⍰ 3. Enum with Custom Values

You can assign custom numeric or string values.

```
tsCopyEditenum Status {
  Success = 200,
  NotFound = 404,
  ServerError = 500,
}

console.log(Status.NotFound); // 404
```

⍰ 4. String Enums

```
tsCopyEditenum Color {
  Red = "RED",
  Green = "GREEN",
  Blue = "BLUE",
}

console.log(Color.Green); // "GREEN"
```

⍰ 5. Reverse Mapping (Only for Numeric Enums)

```
tsCopyEditenum Direction {
  Up,
  Down,
  Left,
  Right,
}

console.log(Direction[0]); // "Up"
```

⍰ Practice Time

1. Create an enum `Role` with values: `Admin = 1`, `User = 2`, `Guest = 3`.
2. Create a string enum `Response` with: `"SUCCESS"`, `"FAILURE"`, `"PENDING"`.
3. Print out values of `Role.Admin` and `Response.FAILURE`.

✓Task Example

```
tsCopyEditenum Role {
  Admin = 1,
  User = 2,
  Guest = 3,
}

enum Response {
  SUCCESS = "SUCCESS",
  FAILURE = "FAILURE",
  PENDING = "PENDING",
}

console.log(Role.Admin);       // 1
console.log(Response.FAILURE); // "FAILURE"
```

Compile and run:

```
bashCopyEdittsc index.ts
node index.js
```

✓Step 12: Generics in TypeScript

⍰ 1. What are Generics?

Generics let you write functions, classes, or interfaces that work with **any type**, while still keeping type safety.

⍰ 2. Generic Function Example

```
tsCopyEditfunction identity<T>(arg: T): T {
  return arg;
}

console.log(identity<string>("Hello"));  // Output: Hello
console.log(identity<number>(42));       // Output: 42
```

- `<T>` is a placeholder for any type.
- When calling, you specify the actual type, e.g., `<string>`, `<number>`.

⍰ 3. Type Inference (No need to specify type explicitly)

```
ts


CopyEdit
console.log(identity("TypeScript")); // TS infers T as string
```

⍰ 4. Generic Array Function

```
tsCopyEditfunction getArray<T>(items: T[]): T[] {
  return new Array().concat(items);
}

const numArray = getArray<number>([1, 2, 3]);
const strArray = getArray<string>(["a", "b", "c"]);

console.log(numArray);
console.log(strArray);
```

⍰ 5. Generic Interface Example

```
tsCopyEditinterface Pair<T, U> {
  first: T;
  second: U;
}

const pair1: Pair<number, string> = { first: 1, second: "one" };
console.log(pair1);
```

⍰ Practice Time

1. Write a generic function `wrapInArray<T>(value: T): T[]` that wraps a value into an array.
2. Create a generic interface `ApiResponse<T>` with `data: T` and `status: number`.

3. Use the interface with a type like `string` or an object.

✓Task Example

```
tsCopyEditfunction wrapInArray<T>(value: T): T[] {
  return [value];
}

console.log(wrapInArray<number>(123));
console.log(wrapInArray<string>("hello"));

interface ApiResponse<T> {
  data: T;
  status: number;
}

const response: ApiResponse<string> = {
  data: "Success!",
  status: 200,
};

console.log(response);
```

Compile & run:

```
bashCopyEdittsc index.ts
node index.js
```

✓Step 13: Classes & Inheritance in TypeScript

▯ 1. Basic Class

```
tsCopyEditclass Person {
  name: string;

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

  greet(): void {
    console.log(`Hello, my name is ${this.name}`);
  }
}

const person1 = new Person("Ankit");
person1.greet(); // Output: Hello, my name is Ankit
```

▯ 2. Class with Default & Optional Values

```
tsCopyEditclass Student {
  name: string;
  age?: number;

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

  info(): void {
    console.log(`Student: ${this.name}, Age: ${this.age ?? "Not given"}`);
  }
}

const s1 = new Student("John");
s1.info();
```

▯ 3. Access Modifiers: public, private, protected

```
tsCopyEditclass Employee {
  public name: string;
  private salary: number;

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

  getDetails(): void {
    console.log(`${this.name}'s salary is ${this.salary}`);
  }
}

const emp = new Employee("Ravi", 50000);
emp.getDetails();
// emp.salary; // ❌Error: private property
```

▯ 4. Inheritance

```
tsCopyEditclass Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }
  move(distance: number) {
    console.log(`${this.name} moved ${distance} meters.`);
  }
}

class Dog extends Animal {
  bark() {
    console.log(`${this.name} says: Woof!`);
  }
}

const d = new Dog("Tommy");
d.bark();
d.move(10);
```

⍰ 5. readonly properties

```
tsCopyEditclass Car {
  readonly model: string;

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

const c1 = new Car("Tesla");
// c1.model = "BMW"; // ❌Cannot assign to 'model' because it is a read-only
property.
```

⍰ Practice Time

1. Create a class `Product` with name and price.
2. Add a method `display()` that prints the product info.
3. Extend `Product` into a `MobilePhone` class with an extra property `brand`.
4. Override the `display()` method in the child class.

✅Example Task

```
tsCopyEditclass Product {
  name: string;
  price: number;

  constructor(name: string, price: number) {
    this.name = name;
    this.price = price;
  }
  display() {
    console.log(`Product: ${this.name}, Price: ${this.price}`);
  }
}

class MobilePhone extends Product {
  brand: string;

  constructor(name: string, price: number, brand: string) {
    super(name, price);
    this.brand = brand;
  }

  display() {
    console.log(`Mobile: ${this.name}, Brand: ${this.brand}, Price:
${this.price}`);
  }
}

const phone = new MobilePhone("iPhone", 70000, "Apple");
phone.display();
```

Compile & run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 14: Interfaces in TypeScript

⍰ 1. What is an Interface?

An **interface** defines the **structure** of an object. It ensures that the object follows a particular shape or rule.

```
tsCopyEditinterface Person {
  name: string;
  age: number;
}

const p1: Person = {
  name: "Ankit",
  age: 21,
};
```

⍰ 2. Optional Properties

```
tsCopyEditinterface Car {
  model: string;
  price?: number; // optional
}

const car1: Car = {
  model: "Tesla",
};
```

⍰ 3. Readonly Properties

```
tsCopyEditinterface User {
  readonly id: number;
  username: string;
}

const u1: User = { id: 1, username: "ankit" };
// u1.id = 2; // ❌Error: Cannot assign to 'id' because it is a read-only
property
```

⍰ 4. Function Types in Interfaces

```
tsCopyEditinterface Greet {
  (name: string): string;
}

const sayHello: Greet = (name) => {
  return `Hello, ${name}`;
};

console.log(sayHello("World"));
```

⍰ 5. Interfaces with Classes

```
tsCopyEditinterface Animal {
  name: string;
  move(): void;
}

class Dog implements Animal {
  name: string;

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

  move() {
    console.log(`${this.name} is running.`);
  }
}

const d1 = new Dog("Tommy");
d1.move();
```

⍰ Practice Time

1. Create an interface `Book` with `title`, `author`, and optional `pages`.
2. Create a function `printBook(book: Book)` that prints details.
3. Create a class `Ebook` that implements `Book` and adds a `fileSize` property.

✓Task Example

```
tsCopyEditinterface Book {
  title: string;
  author: string;
  pages?: number;
}

function printBook(book: Book) {
  console.log(`"${book.title}" by ${book.author} (${book.pages ?? "pages not
given"})`);
}

class Ebook implements Book {
  title: string;
  author: string;
  pages?: number;
  fileSize: number;

  constructor(title: string, author: string, fileSize: number, pages?:
number) {
    this.title = title;
    this.author = author;
    this.fileSize = fileSize;
    this.pages = pages;
  }
}

const myEbook = new Ebook("TypeScript 101", "Ankit", 5, 200);
printBook(myEbook);
```

Compile & run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 15: Type Aliases vs Interfaces

⍰ 1. What is a Type Alias?

A **type alias** gives a name to any type (primitive, union, intersection, etc.)

```
tsCopyEdittype ID = number | string;

let userId: ID = 101;
userId = "abc123";
```

⍰ 2. What is an Interface?

An **interface** defines the structure (shape) of an object or class.

```
tsCopyEditinterface User {
  id: number;
  name: string;
}
```

⍰ 3. When Are They Similar?

You can define object shapes with both:

Using type

```
tsCopyEdittype Person = {
  name: string;
  age: number;
};
```

Using `interface`

```
tsCopyEditinterface Person {
  name: string;
  age: number;
}
```

Both work the same way in most cases.

⍰ 4. Key Differences

Feature	`interface`	`type`
Extendable	✅Can extend other interfaces or classes	✅Can use intersections (&)
Objects	✅Best for object structure	✅Works, but interface is

Feature	`interface`	`type`
		preferred
Primitives / Unions	❌Not allowed	✅Allowed
Declaration merging	✅Supported	❌Not supported

✅Example: Interface vs Type

```
tsCopyEditinterface Animal {
  name: string;
}

interface Dog extends Animal {
  bark(): void;
}

type Cat = Animal & {
  meow(): void;
};
```

▯ 5. Declaration Merging (Only works with interfaces)

```
tsCopyEditinterface Box {
  height: number;
}

interface Box {
  width: number;
}

const b: Box = {
  height: 10,
  width: 5,
}; // Works!
```

With `type`, this would throw an error.

▯ 6. Which One Should You Use?

- Use **`interface`** when you’re working with objects or classes.
- Use **`type`** when you need unions, tuples, or more flexible type expressions.

▯ Practice Time

1. Create a `type` alias for a union of `string` or `number`.

2. Create an `interface` for a `Car` with brand, model, and price.
3. Extend that `Car` interface to create an `ElectricCar` with `batteryLife`.

✅Example Task

```
tsCopyEdittype ID = string | number;

interface Car {
  brand: string;
  model: string;
  price: number;
}

interface ElectricCar extends Car {
  batteryLife: number; // in km
}

const tesla: ElectricCar = {
  brand: "Tesla",
  model: "Model S",
  price: 80000,
  batteryLife: 500,
};

console.log(tesla);
```

Compile & run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 16: Advanced Types in TypeScript

⍰ 1. Union Types (|)

Allows a variable to hold **more than one type**.

```
tsCopyEditlet value: string | number;

value = "hello";
value = 123;
// value = true; // ❌Error
```

Useful when something could be multiple types.

▯ 2. Intersection Types (&)

Combines **multiple types** into one.

```
tsCopyEdittype A = { name: string };
type B = { age: number };

type Person = A & B;

const p: Person = {
  name: "Ankit",
  age: 21,
};
```

▯ 3. Literal Types

Restricts a variable to a **specific set of values**.

```
tsCopyEdittype Direction = "left" | "right" | "up" | "down";

let move: Direction;

move = "left";
// move = "forward"; // ❌Error
```

▯ 4. Type Aliases + Union = Custom Validation

```
tsCopyEdittype Status = "success" | "error" | "loading";

function showStatus(status: Status) {
  console.log(`Current status: ${status}`);
}

showStatus("success");
```

▯ 5. Nullable and Undefined Types

```
tsCopyEditlet username: string | null;
username = null;
username = "ankit";
```

▯ 6. in Operator for Union Discrimination

```
tsCopyEdittype Cat = { meow: () => void };
type Dog = { bark: () => void };
type Animal = Cat | Dog;
```

```
function makeSound(animal: Animal) {
  if ("meow" in animal) {
    animal.meow();
  } else {
    animal.bark();
  }
}
```

⍰ Practice Time

1. Create a union type `Shape = "circle" | "square" | "triangle"`.
2. Write a function that takes shape and logs a message.
3. Create two types: `UserDetails` and `LoginDetails`, and merge them using intersection.

✓Example Task

```
tsCopyEdittype Shape = "circle" | "square" | "triangle";

function draw(shape: Shape) {
  console.log(`Drawing a ${shape}`);
}

draw("square");

type UserDetails = {
  name: string;
  age: number;
};

type LoginDetails = {
  username: string;
  password: string;
};

type FullUser = UserDetails & LoginDetails;

const user: FullUser = {
  name: "Ankit",
  age: 22,
  username: "ankit123",
  password: "secure@pass",
};
```

Compile & run:

```
bashCopyEdittsc index.ts
node index.js
```

✓Step 17: Type Narrowing & Type Guards

⍰ 1. What Is Type Narrowing?

Type narrowing is when TypeScript figures out a more specific type **inside a block of code** based on conditions.

```
tsCopyEditfunction printId(id: string | number) {
  if (typeof id === "string") {
    console.log(id.toUpperCase()); // Now TypeScript knows it's a string
  } else {
    console.log(id.toFixed(2)); // Now it's a number
  }
}
```

⍰ 2. typeof Narrowing

Works for primitive types like `string`, `number`, `boolean`, `undefined`.

```
tsCopyEditfunction handleInput(input: string | boolean) {
  if (typeof input === "string") {
    console.log("You typed: " + input);
  } else {
    console.log("Boolean value: " + input);
  }
}
```

⍰ 3. in Operator Narrowing

Used to detect if a property **exists** in an object type.

```
tsCopyEdittype Dog = { bark: () => void };
type Cat = { meow: () => void };

function makeSound(animal: Dog | Cat) {
  if ("bark" in animal) {
    animal.bark();
  } else {
    animal.meow();
  }
}
```

⍰ 4. `instanceof` Narrowing

Used to check if a value is an **instance of a class**.

```
tsCopyEditclass Car {
  drive() {
    console.log("Driving...");
  }
}

class Bike {
  ride() {
    console.log("Riding...");
  }
}

function useVehicle(vehicle: Car | Bike) {
  if (vehicle instanceof Car) {
    vehicle.drive();
  } else {
    vehicle.ride();
  }
}
```

⍰ 5. Custom Type Guards

Create your own function to narrow types.

```
tsCopyEdittype Fish = { swim: () => void };
type Bird = { fly: () => void };

function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}

function move(pet: Fish | Bird) {
  if (isFish(pet)) {
    pet.swim();
  } else {
    pet.fly();
  }
}
```

⍰ Practice Time

1. Create a function `display()` that takes a `string | number | boolean`.
2. Use `typeof` to narrow the type and print a specific message for each.

3. Create two types `Admin` and `Guest`, and use a custom type guard to check if user is admin.

✅Example Task

```
tsCopyEditfunction display(val: string | number | boolean) {
  if (typeof val === "string") {
    console.log("Text: " + val.toUpperCase());
  } else if (typeof val === "number") {
    console.log("Number: " + val.toFixed(1));
  } else {
    console.log("Boolean: " + (val ? "Yes" : "No"));
  }
}

type Admin = {
  role: "admin";
  accessLevel: number;
};

type Guest = {
  role: "guest";
};

function isAdmin(user: Admin | Guest): user is Admin {
  return (user as Admin).accessLevel !== undefined;
}

function checkAccess(user: Admin | Guest) {
  if (isAdmin(user)) {
    console.log("Access level:", user.accessLevel);
  } else {
    console.log("Guest access");
  }
}
```

Compile & run:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 18: Generics in TypeScript

□ 1. What Are Generics?

Generics allow you to **define types dynamically** — instead of hardcoding a specific type, you use a **placeholder**.

```
tsCopyEditfunction identity<T>(arg: T): T {
  return arg;
}

const output1 = identity<string>("Hello");
const output2 = identity<number>(100);
```

▯ T is a **type variable**, which gets replaced by the actual type you pass.

▯ 2. Why Use Generics?

- Reusable: No need to write separate versions for each type.
- Type-Safe: TypeScript still knows what type is used.
- Flexible: Works with any type.

▯ 3. Generic Functions

```
tsCopyEditfunction wrapInArray<T>(value: T): T[] {
  return [value];
}

const result = wrapInArray("hello"); // string[]
```

▯ 4. Generic Interfaces

```
tsCopyEditinterface Box<T> {
  value: T;
}

const stringBox: Box<string> = { value: "Book" };
const numberBox: Box<number> = { value: 123 };
```

▯ 5. Generic Classes

```
tsCopyEditclass DataStore<T> {
  private data: T[] = [];

  addItem(item: T) {
    this.data.push(item);
  }

  getItems(): T[] {
    return this.data;
  }
}
```

```
const stringStore = new DataStore<string>();
stringStore.addItem("Ankit");

const numberStore = new DataStore<number>();
numberStore.addItem(10);
```

⍰ 6. Generic Constraints (extends)

You can **limit** what type `T` can be.

```
tsCopyEditfunction printLength<T extends { length: number }>(item: T) {
  console.log(item.length);
}

printLength("hello"); // ✅
printLength([1, 2, 3]); // ✅
// printLength(100); // ❌Error: number has no length
```

⍰ 7. Default Generic Type

```
tsCopyEditfunction makePair<T = string, U = number>(key: T, value: U) {
  return { key, value };
}

const p = makePair("id", 101);       // T=string, U=number
const p2 = makePair(undefined, 42);  // T=string (default)
```

⍰ Practice Task

1. Create a function `merge<T, U>(obj1: T, obj2: U)` that returns one object with all keys.
2. Create a `Stack<T>` class that can `push()`, `pop()` and `getAll()` items.

✅Example Task

```
tsCopyEditfunction merge<T, U>(obj1: T, obj2: U): T & U {
  return { ...obj1, ...obj2 };
}

const merged = merge({ name: "Ankit" }, { age: 22 });
console.log(merged);

class Stack<T> {
  private items: T[] = [];

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

  pop(): T | undefined {
    return this.items.pop();
  }

  getAll(): T[] {
    return this.items;
  }
}

const numberStack = new Stack<number>();
numberStack.push(10);
numberStack.push(20);
console.log(numberStack.getAll());
```

Run with:

```
bashCopyEdittsc index.ts
node index.js
```

✅Step 19: Utility Types in TypeScript

⍰ 1. `Partial<Type>`

Makes **all properties optional**.

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

const updateUser = (user: Partial<User>) => {
  // You can pass only part of the user
  console.log(user);
};

updateUser({ name: "Ankit" }); // age is not required
```

⍰ 2. `Required<Type>`

Makes **all properties required**, even if they were optional.

```
tsCopyEdittype User = {
  name?: string;
  age?: number;
};
```

```
const u: Required<User> = {
  name: "Ankit",
  age: 22
};
```

⍰ 3. Readonly<Type>

Makes **all properties read-only** (can’t change them).

```
tsCopyEdittype User = {
  name: string;
};

const u: Readonly<User> = {
  name: "Ankit"
};

// u.name = "Raj"; ❌Error: Cannot assign to 'name'
```

⍰ 4. Pick<Type, Keys>

Selects only the specified keys from a type.

```
tsCopyEdittype User = {
  name: string;
  age: number;
  email: string;
};

type UserNameAndEmail = Pick<User, "name" | "email">;

const user: UserNameAndEmail = {
  name: "Ankit",
  email: "a@example.com"
};
```

⍰ 5. Omit<Type, Keys>

Removes specified keys from a type.

```
tsCopyEdittype User = {
  name: string;
  age: number;
  email: string;
};
```

```
type UserWithoutEmail = Omit<User, "email">;

const user: UserWithoutEmail = {
  name: "Ankit",
  age: 22
};
```

⍰ 6. `Record<Keys, Type>`

Creates a type with **specific keys** and one value type.

```
tsCopyEdittype Role = "admin" | "user" | "guest";

const roles: Record<Role, string> = {
  admin: "Admin User",
  user: "Regular User",
  guest: "Guest User"
};
```

⍰ 7. `Exclude<UnionType, ExcludedMembers>`

Removes some members from a union type.

```
tsCopyEdittype Status = "active" | "inactive" | "banned";

type VisibleStatus = Exclude<Status, "banned">;
// Result: "active" | "inactive"
```

⍰ Practice Task

1. Create a `Product` type with `name`, `price`, `category`.
2. Create:
   - `Partial<Product>` for update function
   - `Readonly<Product>` for config data
   - `Pick<Product, "name" | "price">` for public view

✓Example Task

```
tsCopyEdittype Product = {
  name: string;
  price: number;
  category: string;
};
```

```
function updateProduct(p: Partial<Product>) {
  console.log("Updated:", p);
}

const productConfig: Readonly<Product> = {
  name: "Laptop",
  price: 999,
  category: "Electronics"
};

const publicProduct: Pick<Product, "name" | "price"> = {
  name: "Laptop",
  price: 999
};
```

TypeScript utility types help you **save time**, reduce bugs, and keep code consistent.

✓Step 20: Declaration Merging in TypeScript

□ What is Declaration Merging?

Declaration Merging means that **TypeScript automatically combines multiple declarations** of the same **interface**, **function**, **enum**, or **namespace** into one.

□ 1. Merging Interfaces

You can declare an interface more than once, and TypeScript will merge them together.

```
tsCopyEditinterface Person {
  name: string;
}

interface Person {
  age: number;
}

const p: Person = {
  name: "Ankit",
  age: 22
};
```

□ This is **common in type definitions**, especially in libraries.

▯ 2. Merging Functions + Namespaces

You can merge a **function and a namespace** to add extra functionality.

```
tsCopyEditfunction greet(name: string) {
  return `Hello, ${name}`;
}

namespace greet {
  export const version = "1.0";
}

console.log(greet("Ankit"));       // Hello, Ankit
console.log(greet.version);        // 1.0
```

▯ 3. Merging Enums

Only string or numeric enums can merge.

```
tsCopyEditenum Status {
  Active = "ACTIVE"
}

enum Status {
  Inactive = "INACTIVE"
}

console.log(Status.Active);    // ACTIVE
console.log(Status.Inactive);  // INACTIVE
```

▯ Rules & Warnings

- Only certain things like **interfaces**, **enums**, **functions with namespaces** can merge.
- If two declarations **conflict** (e.g., same property with different types), it will throw an error.
- **Classes cannot be merged** like interfaces.

▯ Practice Task

1. Create two `interface` declarations for `Car`:
   - First with `brand` and `model`
   - Second with `year` and `price`
2. Create a variable of type `Car` with all 4 properties.

✅Example Task

```
tsCopyEditinterface Car {
  brand: string;
  model: string;
}

interface Car {
  year: number;
  price: number;
}

const myCar: Car = {
  brand: "Toyota",
  model: "Camry",
  year: 2023,
  price: 25000
};
```

Declaration Merging is especially helpful when working with **third-party libraries** like `express` or `React` where you extend existing types/interfaces.

✅Step 21: Modules & Namespaces in TypeScript

(Organizing Code in Larger Projects)

⍰ What Are Modules?

A **module** is any file in TypeScript that **exports or imports** something.

⍰ When you split code across multiple files and use `import/export`, you're using modules.

⍰ Example: Creating and Using Modules

`mathUtils.ts`

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

export const PI = 3.14;
```

`main.ts`

```
tsCopyEditimport { add, PI } from "./mathUtils";

console.log(add(2, 3));     // 5
console.log(PI);            // 3.14
```

> ⍰ Tip: File paths must be correct (`./`, `../`), and TypeScript must be configured to support modules (`tsconfig.json` should have `"module": "ESNext"` or `"CommonJS"`).

⍰ `export default`

You can export **only one default** from a file.

`user.ts`

```
tsCopyEditexport default function greet(name: string) {
  return `Hello, ${name}`;
}
```

`main.ts`

```
tsCopyEditimport greet from "./user";

console.log(greet("Ankit")); // Hello, Ankit
```

⍰ Namespaces (Old Way)

Before ES Modules, TypeScript used **`namespace`** to group code.

Example:

```
tsCopyEditnamespace Geometry {
  export function areaOfSquare(side: number): number {
    return side * side;
  }
}

console.log(Geometry.areaOfSquare(5)); // 25
```

> ⍰ **Namespaces are not used in modern TypeScript** if you're using ES Modules. Use them only for old codebases or internal projects.

✓When to Use Modules?

- When organizing code in **multiple files**
- For **reusability**, **scalability**, and **clarity**
- With **Node.js**, **React**, **Angular**, **Vue**, etc.

⍰ Practice Task

1. Create a file `stringUtils.ts` with two exports:
   - `toUpperCase(str: string)`
   - `toLowerCase(str: string)`

2. Import and use these in `main.ts`.

⍰ Directory Example:

```
cssCopyEditproject/
│
├── stringUtils.ts
├── main.ts
└── tsconfig.json
```

⍰ Compile & Run (for modules):

```
bashCopyEdittsc       # compiles all .ts files based on tsconfig.json
node main.js
```

That finishes **Phase 3: TypeScript Core Deep Dive**! ⍰ Here's a quick recap of what you've mastered in Phase 3:

✅Advanced Types ✅Type Narrowing ✅Literal & Template Types ✅Type Guards ✅Utility Types ✅Declaration Merging ✅Modules & Namespaces

✅Step 22: Classes in TypeScript

⍰ What is a Class?

A **class** is like a blueprint to create objects with properties and methods.

⍰ Basic Class Syntax

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

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

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

const p = new Person("Ankit", 22);
p.greet();  // Hello, my name is Ankit
```

? Important Points:

- `constructor` is a special method for initializing new objects.
- Properties like `name` and `age` can be declared inside the class.
- Methods like `greet()` belong to the class.

? Shorter Property Declaration in Constructor

TypeScript lets you declare and initialize properties in the constructor parameters directly:

```
tsCopyEditclass Person {
  constructor(public name: string, public age: number) {}

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

const p = new Person("Ankit", 22);
p.greet();
```

? Practice Task:

1. Create a class `Car` with properties `brand`, `model`, and `year`.
2. Add a method `getDetails()` that prints all the details.
3. Create an instance and call the method.

✓Step 23: Inheritance and `super` Keyword in TypeScript

? What is Inheritance?

Inheritance lets a **child class** reuse or extend the behavior of a **parent class** — so you don't have to rewrite common code.

? Basic Example

```
tsCopyEditclass Animal {
  constructor(public name: string) {}

  move(distance: number) {
    console.log(`${this.name} moved ${distance} meters.`);
  }
}
```

```
class Dog extends Animal {
  bark() {
    console.log("Woof! Woof!");
  }
}

const dog = new Dog("Buddy");
dog.bark();               // Woof! Woof!
dog.move(10);             // Buddy moved 10 meters.
```

⍰ Using super

The super keyword calls the parent class constructor or methods.

```
tsCopyEditclass Animal {
  constructor(public name: string) {}

  move(distance: number) {
    console.log(`${this.name} moved ${distance} meters.`);
  }
}

class Dog extends Animal {
  constructor(name: string, public breed: string) {
    super(name);  // Calls Animal's constructor
  }

  bark() {
    console.log(`${this.name} says: Woof! Woof!`);
  }
}

const dog = new Dog("Buddy", "Golden Retriever");
dog.bark();               // Buddy says: Woof! Woof!
dog.move(15);             // Buddy moved 15 meters.
```

⍰ Method Overriding

Child classes can **override** parent methods:

```
tsCopyEditclass Dog extends Animal {
  move(distance: number) {
    console.log("Dog is running...");
    super.move(distance);  // Call parent method too
  }
}
```

Practice Task

1. Create a parent class `Vehicle` with `make` and `speed` properties and a method `move()`.
2. Create a child class `Car` that extends `Vehicle` and adds a method `honk()`.
3. Override `move()` in `Car` to add a custom message before calling parent `move()`.

✅Step 24: Access Modifiers in TypeScript — `public`, `private`, and `protected`

What Are Access Modifiers?

They control **visibility and accessibility** of class members (properties and methods).

Types of Access Modifiers

Modifier	Accessible Outside Class?	Accessible in Subclass?	Accessible in Same Class?
`public`	Yes	Yes	Yes
`private`	No	No	Yes
`protected`	No	Yes	Yes

Examples

```
tsCopyEditclass Person {
  public name: string;          // accessible anywhere
  private ssn: string;          // accessible only inside Person
  protected age: number;        // accessible inside Person & subclasses

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

  public showInfo() {
    console.log(`Name: ${this.name}, Age: ${this.age}`);
  }

  private getSSN() {
    return this.ssn;
  }
}

class Employee extends Person {
  constructor(name: string, ssn: string, age: number, public jobTitle:
string) {
    super(name, ssn, age);
  }

  public showDetails() {
    console.log(`Job Title: ${this.jobTitle}, Age: ${this.age}`); // can
access protected age
  }
}

const p = new Person("Ankit", "123-45-6789", 22);
console.log(p.name);       // OK
// console.log(p.ssn);     // Error: private property
// console.log(p.age);     // Error: protected property

const e = new Employee("John", "987-65-4321", 30, "Developer");
e.showDetails();           // OK
```

⍰ Why Use Access Modifiers?

- **Encapsulation:** Hide sensitive data (`private`).
- **Controlled exposure:** Let subclasses access some data (`protected`).
- **Open access:** Use `public` for everything else.

⍰ Practice Task:

1. Create a `BankAccount` class:
   - `public accountHolder: string`
   - `private balance: number`
   - `protected accountNumber: string`
2. Add methods:
   - `deposit(amount: number)` to add money.
   - `getBalance()` to return balance (only inside class).
3. Extend with a subclass `SavingsAccount`:
   - Access `accountNumber` in a method `showAccountNumber()`.

✓Step 25: Abstract Classes and Methods in TypeScript

⍰ What Are Abstract Classes?

- Abstract classes **can't be instantiated** directly.
- They are **blueprints** for other classes.
- Can contain **abstract methods** that **must be implemented** by subclasses.

⍰ Syntax Example

```
tsCopyEditabstract class Animal {
  constructor(public name: string) {}

  abstract makeSound(): void;  // Abstract method, no body

  move() {
    console.log(`${this.name} is moving.`);
  }
}

class Dog extends Animal {
  makeSound() {
    console.log("Woof! Woof!");
  }
}

const dog = new Dog("Buddy");
dog.move();       // Buddy is moving.
dog.makeSound();  // Woof! Woof!

// const animal = new Animal("Generic"); // Error: Cannot create an instance
of an abstract class.
```

⍰ Key Points:

- Abstract class **can have concrete methods** (like `move`).
- Abstract methods **must be implemented** in subclasses.
- Abstract classes **cannot be instantiated** directly.

⍰ Practice Task:

1. Create an abstract class `Shape` with:
   - An abstract method `area(): number`.
   - A concrete method `describe()` that logs `"I am a shape."`.
2. Create subclasses `Circle` and `Rectangle`:
   - Implement the `area()` method for each shape.
3. Create objects of `Circle` and `Rectangle`, call `area()` and `describe()`.

Step 26: Interfaces vs Abstract Classes in TypeScript

⍰ What is an Interface?

- An **interface** defines the **shape** of an object (properties & methods) without implementation.

- It’s a **contract** that a class or object must follow.

⍰ Example Interface:

```
tsCopyEditinterface Movable {
  speed: number;
  move(): void;
}

class Car implements Movable {
  constructor(public speed: number) {}

  move() {
    console.log(`Moving at speed: ${this.speed}`);
  }
}
```

⍰ Key Differences Between Interfaces & Abstract Classes

Feature	Interface	Abstract Class
Can contain method bodies	No (only in TS 4.3+ default methods)	Yes
Can have constructors	No	Yes
Can implement multiple	Yes (multiple inheritance)	No (single inheritance)
Supports properties	Yes	Yes
Use for	Defining shapes/contracts	Providing base classes with some implementation
Can be instantiated	No	No

⍰ When to Use What?

- Use **interfaces** to define **shapes and contracts**.
- Use **abstract classes** when you want to share **common behavior/code** between subclasses.

⍰ Practice Task:

1. Create an interface `Printable` with a method `print(): void`.
2. Create an abstract class `Document` with a method `open()` and an abstract method `print()`.
3. Create classes `PDFDocument` and `WordDocument` that:
   - Extend `Document`.

  - Implement `Printable`.
  - Provide their own `print()` methods.

✓Step 27: Modules in TypeScript

⍰ What Are Modules?

- Modules help you **split your code into separate files**.
- Each module can **export** variables, functions, classes, interfaces, etc.
- Other modules can **import** these exported members to use them.
- This keeps code **clean, reusable, and maintainable**.

⍰ Why Use Modules?

- Avoid polluting the global scope.
- Organize related code.
- Enable code reuse across files.
- Support encapsulation.

⍰ How to Export and Import

---

1. Named Exports and Imports

File: `mathUtils.ts`

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

export const PI = 3.1416;
```

File: `app.ts`

```
tsCopyEditimport { add, PI } from './mathUtils';

console.log(add(5, 3)); // 8
console.log(PI);        // 3.1416
```

---

2. Default Export and Import

File: `logger.ts`

```
tsCopyEditexport default function log(message: string): void {
  console.log("LOG:", message);
}
```

File: `app.ts`

```
tsCopyEditimport log from './logger';

log("This is a default export example.");
```

Importing Everything as an Object

```
tsCopyEditimport * as math from './mathUtils';

console.log(math.add(4, 6));  // 10
console.log(math.PI);          // 3.1416
```

Module Resolution & Running Modules in VS Code

- TypeScript modules use ES module syntax.
- To run your TypeScript files that use modules, **compile** them first:

  ```
  bash


  CopyEdit
  tsc
  ```

- Then run the compiled JavaScript with Node.js:

  ```
  bash


  CopyEdit
  node dist/app.js
  ```

- Or use tools like **`ts-node`** to run TypeScript directly:

  ```
  bash


  CopyEdit
  npx ts-node app.ts
  ```

Practice Task:

1. Create a file `calculator.ts` with named exports for functions: `add`, `subtract`, `multiply`, `divide`.
2. Create another file `main.ts` to import and use these functions.
3. Use both named imports and import everything as an object.

4. Try default export by creating a `greet.ts` file exporting a default function that logs a greeting.

✓Step 28: Namespaces vs Modules in TypeScript

⍰ What Are Namespaces?

- **Namespaces** are a TypeScript-specific way to group related code under a single global name.
- They use the `namespace` keyword.
- Mainly useful in **older codebases** or when you want to avoid polluting the global scope **without using modules**.
- Can be split across multiple files using `///<reference>` directives, but generally less used now.

⍰ What Are Modules?

- Modules are based on **ES6 modules** (`import`/`export` syntax).
- Each file is a module.
- The preferred way to organize code in modern TypeScript.
- Modules always have their own scope — no risk of global pollution.

⍰ Key Differences

Feature	Namespace	Module
Declaration	`namespace MyNamespace { }`	Use `export` / `import` in files
Scope	Adds to global scope (unless nested)	File-scoped, no global scope pollution
Module system	TypeScript only	Follows ES6 module standard
Loading	Synchronous (at compile time)	Asynchronous (runtime loading with bundlers/Node.js)
Use case	Large legacy projects, internal grouping	Modern apps, external modules, libraries

---

⍰ Namespace Example

```
tsCopyEditnamespace Utility {
  export function greet(name: string) {
    console.log(`Hello, ${name}!`);
  }
}

Utility.greet("Ankit");  // Hello, Ankit!
```

⍰ Module Example

File: `utils.ts`

```
tsCopyEditexport function greet(name: string) {
  console.log(`Hello, ${name}!`);
}
```

File: `app.ts`

```
tsCopyEditimport { greet } from './utils';

greet("Ankit");
```

⍰ When To Use What?

- Use **Modules** for modern, scalable applications and libraries.
- Use **Namespaces** when:
  - You work in a legacy codebase without module loaders.
  - You want to group related code internally without external dependencies.

⍰ Practice Task

1. Create a namespace `MathUtils` with functions `add` and `multiply`.
2. Call those functions outside the namespace.
3. Create two separate files with modules exporting functions and import them in a main file.
4. Observe the scope and behavior differences.