


Assume you are good at problem solving in typescript and good guide for beginner then provide best examples for  Static Typing Type Annotations Declare types for variables, function parameters, return values, etc. Example: let count: number = 5;

Absolutely, Routh Kiran! Let's walk through **Static Typing** and **Type Annotations** in **TypeScript** with well-structured and beginner-friendly examples. These examples will help you understand how to declare types for:

- Variables
- Function Parameters
- Return Values
- Arrays
- Objects
- Union Types
- Custom Types / Interfaces

---

## ✓ 1. Static Typing with Variable Declaration

```
let count: number = 5;
let username: string = "Routh";
let isLoggedIn: boolean = true;
```

---

## ✓ 2. Function Parameters with Type Annotations

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

```
greet("Kiran"); // Output: Hello, Kiran!
```

---

### ✓ 3. Function Return Type Annotation

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

```
let sum = add(10, 20); // sum: number = 30
```

---

### ✓ 4. Arrays with Type Annotation

```
let numbers: number[] = [1, 2, 3, 4, 5];  
let fruits: string[] = ["apple", "banana", "mango"];
```

---

### ✓ 5. Objects with Type Annotations

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

---

### ✓ 6. Type Aliases (Custom Types)

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

```
let newUser: User = {  
    name: "Routh",  
    email: "routh@example.com",  
    isActive: true  
};
```

---

## ✓ 7. Interfaces (Preferred for Objects)

```
interface Product {  
  id: number;  
  name: string;  
  price: number;  
}
```

```
let item: Product = {  
  id: 101,  
  name: "Keyboard",  
  price: 699  
};
```

---

## ✓ 8. Function with Object Parameter and Return Type

```
interface Book {  
  title: string;  
  author: string;  
}
```

```
function getBookTitle(book: Book): string {  
  return book.title;  
}
```

```
let myBook: Book = {  
  title: "Atomic Habits",  
  author: "James Clear"  
};
```

```
console.log(getBookTitle(myBook)); // Output: Atomic Habits
```

---

## ✓ 9. Union Types

Allow a variable to hold more than one type.

```
let id: number | string;
```

```
id = 101;  
id = "ABC123";
```

---

## ✓ 10. Optional Properties and Default Values

```
function logMessage(message: string, userId?: number): void {  
  console.log(`Message: ${message}`);  
  if (userId !== undefined) {  
    console.log(`User ID: ${userId}`);  
  }  
}
```

```
logMessage("Hello World");  
logMessage("Welcome", 42);
```

---

## ✓ 11. Readonly Types

```
interface Config {  
  readonly appName: string;  
  version: number;  
}
```

```
let config: Config = {  
  appName: "MyApp",  
  version: 1.0  
};
```

// config.appName = "AnotherApp"; ❌ Error: Cannot assign to 'appName' because it is a read-only property

---


## ■ Summary Table

Feature	Syntax Example
Variable Type	<code>let x: number = 10;</code>
Function Params	<code>function greet(name: string)</code>
Return Type	<code>function add(a: number): number</code>
Array Type	<code>let nums: number[] = [1, 2, 3];</code>
Object Type	<code>let user: { name: string; age: number }</code>

Union Type	<code>`let id: number</code>
Interface	<code>interface User { name: string }</code>
Type Alias	<code>type User = { ... }</code>

---

Would you like a small project idea that combines all these? Or a playground link to try them online?

Assume you are good at problem solving in typescript and good guide for beginner from beginner to advance then provide best examples for  Interfaces & Types interface, type Define custom type shapes for objects, functions, etc.  
Example: interface User { name: string; age: number }

Absolutely, Routh! Let's dive deep into **Interfaces and Types** in **TypeScript**, especially for beginners progressing to advanced level. I'll walk you through the fundamentals and build up with meaningful examples.

---

## ◆ 1. What are Interfaces & Types in TypeScript?

TypeScript allows us to define **custom data structures** using `interface` and `type`. These help us:

- Describe the **shape of objects**
  - **Enforce structure**
  - Catch **type errors at compile time**
-

## ◆ 2. Basic Syntax Comparison

```
// Interface
interface User {
  name: string;
  age: number;
}
```

```
// Type
type Product = {
  id: number;
  title: string;
};
```

Both work similarly for object structures but have key differences you'll see as we go further.

---



## SECTION 1: BEGINNER LEVEL



### Example 1: Simple Object Shape

```
interface User {
  name: string;
  age: number;
}
```

```
const user1: User = {
  name: "Alice",
  age: 25,
};
```

```
type Product = {
  id: number;
  price: number;
};
```

```
const item: Product = {
  id: 1,
  price: 199.99,
};
```

---

### ✓ Example 2: Optional & Readonly Properties

```
interface Profile {  
  username: string;  
  bio?: string; // optional  
  readonly id: number;  
}
```

```
const p1: Profile = { username: "Kiran", id: 101 };  
// p1.id = 200; ✗ Error: Cannot assign to 'id' because it is a read-only property
```

---

### ✓ Example 3: Interface for Function Type

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

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

```
type Add = (a: number, b: number) => number;
```

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

---

## 🧱 SECTION 2: INTERMEDIATE LEVEL

### ✓ Example 4: Extending Interfaces & Types

```
interface Animal {  
  name: string;  
}
```

```
interface Dog extends Animal {  
  breed: string;  
}
```

```
const d: Dog = {  
  name: "Buddy",  
  breed: "Labrador",  
};
```

```
type Shape = {
```

```
    color: string;
};

type Circle = Shape & {
    radius: number;
};

const c: Circle = { color: "red", radius: 10 };
```

---

### ✓ Example 5: Union & Intersection Types

```
type Admin = {
    role: string;
};

type Person = {
    name: string;
};

type AdminPerson = Admin & Person;

const admin: AdminPerson = {
    name: "Kiran",
    role: "Manager",
};

type Status = "active" | "inactive" | "pending";

let currentStatus: Status = "active";
```

---

### ✓ Example 6: Interface for Class Structure

```
interface Vehicle {
    start(): void;
    stop(): void;
}

class Car implements Vehicle {
    start() {
        console.log("Car started");
    }
}
```



```
stop() {  
  console.log("Car stopped");  
}  
}
```

---

## SECTION 3: ADVANCED LEVEL

### Example 7: Generics with Interfaces and Types

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

```
const response1: ApiResponse<string> = {  
  status: 200,  
  data: "Success",  
};
```

```
const response2: ApiResponse<number[]> = {  
  status: 200,  
  data: [1, 2, 3],  
};
```

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

```
const coordinates: Pair<number, number> = { first: 10, second: 20 };
```

---

### Example 8: Discriminated Unions (for Type Narrowing)

```
type Square = { kind: "square"; size: number };  
type Rectangle = { kind: "rectangle"; width: number; height: number };  
type Shape = Square | Rectangle;
```

```
function area(shape: Shape): number {  
  if (shape.kind === "square") return shape.size * shape.size;  
  else return shape.width * shape.height;
```

```
}
```

---

### ✅ Example 9: Mapped Types (Advanced Types)

```
type ReadOnly<T> = {  
  readonly [P in keyof T]: T[P];  
};
```

```
type UserInfo = {  
  name: string;  
  email: string;  
};
```

```
type ReadOnlyUserInfo = ReadOnly<UserInfo>;  
// All properties in ReadOnlyUserInfo are now readonly
```

---

### ✅ Example 10: Conditional Types

```
type IsString<T> = T extends string ? "Yes" : "No";
```

```
type A = IsString<string>; // "Yes"  
type B = IsString<number>; // "No"
```

---

## 🧠 WHEN TO USE **interface** vs **type**?

Feature	interface ✅	type ✅
Object shapes	✅	✅
Function types	✅	✅
Extending	✅ (using <b>extends</b> )	✅ (with <b>&amp;</b> )
Union types	❌	✅
Intersection types	❌	✅

Declaration  
merging



✓ **Use `interface`** when you are defining a structure to be implemented or extended (like OOP).

✓ **Use `type`** when combining multiple types, using unions, tuples, or advanced type manipulation.



## QUICK PRACTICE IDEAS

1. **Create a `UserProfile` interface** with optional address and a method to display full name.
2. **Make a `type` called `ApiResponse<T>`** and test it with various generics.
3. **Build a `Shape` union type** and write a function to return the area of a shape.
4. **Define a `readonly` mapped type** to protect object mutation.

Would you like these examples in a downloadable file or turned into a TypeScript playground link for hands-on coding?



**Generics      Generics      Create reusable components that work with multiple types. Example:**  
**`function identity<T>(arg: T): T { return arg; }`**

Absolutely, Routh Kiran Babu! Let's go from **beginner to advanced** step-by-step to deeply understand **Generics** in **TypeScript** with clear, real-world examples.



## What Are Generics in TypeScript?

Generics allow us to **create reusable and flexible functions, classes, and interfaces** that can work with **any data type**, without losing **type safety**.

---

## LEVEL 1: Basic Generics – Understand the Core Concept

### ♦ Example 1: Identity Function

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

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

**Why it's useful:** You don't have to write separate functions for `string`, `number`, `boolean`, etc.

---

## LEVEL 2: Generic Arrays and Type Inference

### ♦ Example 2: Work with Arrays

```
function firstElement<T>(arr: T[]): T {  
  return arr[0];  
}
```

```
console.log(firstElement([10, 20, 30])); // Output: 10  
console.log(firstElement(["a", "b", "c"])); // Output: "a"
```

### ♦ Example 3: Type Inference without Explicit `<T>`

```
function logAndReturn<T>(arg: T): T {  
  console.log(typeof arg);  
}
```

```
    return arg;
}
```

```
let result = logAndReturn("TypeScript"); // inferred as string
```

---

## LEVEL 3: Generics with Interfaces

### ◆ Example 4: Generic Interface for API Response

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

```
const userResponse: ApiResponse<{ name: string; age: number }> = {
    status: 200,
    data: {
        name: "Kiran",
        age: 25
    }
};
```

---

## LEVEL 4: Generics with Classes

### ◆ Example 5: Generic Class

```
class Box<T> {
    private value: T;

    constructor(value: T) {
        this.value = value;
    }

    getValue(): T {
        return this.value;
    }
}
```

```
const numberBox = new Box<number>(100);
const stringBox = new Box<string>("Tools");
```

```
console.log(numberBox.getValue()); // 100
console.log(stringBox.getValue()); // Tools
```

---

## LEVEL 5: Generic Constraints – Restricting Type Parameters

### ◆ Example 6: Constrain to Objects with **length**

```
function logLength<T extends { length: number }>(arg: T): T {
  console.log("Length:", arg.length);
  return arg;
}
```

```
logLength("Kiran");
logLength([1, 2, 3]);
// logLength(123); ❌ Error: number doesn't have 'length'
```

---

## LEVEL 6: Generics with Multiple Type Parameters

### ◆ Example 7: Merge Two Objects

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

```
const merged = merge({ name: "Kiran" }, { age: 25 });
console.log(merged.name); // Kiran
console.log(merged.age); // 25
```

---

## LEVEL 7: Default Generic Type

### ◆ Example 8: Provide Default Types

```
function makePair<T = string, U = number>(first: T, second: U): [T, U] {  
  return [first, second];  
}
```

```
const pair1 = makePair("Hello", 100); // [string, number]  
const pair2 = makePair(undefined, 200); // [string, number]
```

---

## LEVEL 8: Keyof and Generics (Type Mapping)

### ◆ Example 9: Get Property Value from Object Safely

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

```
const person = { name: "Kiran", age: 25 };  
const name = getProp(person, "name"); // Valid  
// const invalid = getProp(person, "height"); ❌
```

---

## LEVEL 9: Generic Utility Types

### ◆ Example 10: Partial, Readonly, etc.

```
interface User {  
  name: string;  
  age: number;  
}
```

```
const partialUser: Partial<User> = { name: "Kiran" };  
const readonlyUser: Readonly<User> = { name: "Kiran", age: 25 };  
// readonlyUser.age = 30; ❌ Not allowed
```

---

## LEVEL 10: Advanced – Generic Factory Function

### ♦ Example 11: Create Instances of a Class

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

```
function createInstance<T>(c: { new (): T }): T {  
  return new c();  
}
```

```
const myCar = createInstance(Car);  
myCar.drive(); // Driving...
```

---

### Summary Table

Level	Topic	Example
1	Basic Generic Function	<code>identity&lt;T&gt;(arg: T): T</code>
2	Generic Array Function	<code>firstElement&lt;T&gt;(arr: T[])</code>
3	Type Inference	<code>logAndReturn("Hello")</code>
4	Generic Interface	<code>ApiResponse&lt;T&gt;</code>
5	Generic Class	<code>class Box&lt;T&gt;</code>
6	Generic Constraints	<code>T extends { length: number }</code>
7	Multiple Generics	<code>merge&lt;T, U&gt;(...)</code>



8	Default Generics	<code>makePair&lt;T = string, U = number&gt;</code>
9	Keyof + Generic	<code>getProp&lt;T, K extends keyof T&gt;</code>
10	Utility Types (Partial, etc.)	<code>Partial&lt;T&gt;, Readonly&lt;T&gt;</code>
11	Generic Factory	<code>createInstance&lt;T&gt;(...)</code>

---

If you want, I can make this a **PDF**, or help you build a **TypeScript notebook** project to practice these interactively.

Would you like that?



## Type Safety Type Inference, Type Guards

TypeScript can infer types or narrow them down using logic (like `typeof`, `instanceof`).

Absolutely, Kiran! Let's dive deep into **Type Safety**, **Type Inference**, and **Type Guards** in **TypeScript** with beginner-to-advanced **examples and explanations**. By the end, you'll understand how TypeScript helps catch errors at compile time and makes your code **safe, readable, and predictable**.

---



## 1. Type Safety

Type safety ensures variables, functions, and objects follow strict typing rules.

### ✓ Basic Example (Beginner)

```
let age: number = 25;
// age = "twenty-five"; ❌ Error: Type 'string' is not assignable to type 'number'
```

### ✓ Function Example

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

```
console.log(greet("Kiran")); // ✓  
console.log(greet(123));    // ✗ Error: Argument of type 'number' is not assignable to  
parameter of type 'string'
```

---

## 2. Type Inference

TypeScript **infers types automatically** when not explicitly declared.

### ✓ Basic Inference (Beginner)

```
let country = "India"; // inferred as string  
country = "USA";      // ✓  
country = 91;         // ✗ Error: number not assignable to string
```

### ✓ Function Return Type Inference

```
function add(a: number, b: number) {  
  return a + b; // inferred as number  
}  
  
const result = add(5, 10); // result is of type number
```

---

## 3. Type Guards

Type guards **narrow down** types using runtime checks like `typeof`, `instanceof`, or custom logic.

---

### ✓ a) `typeof` Guard (Strings, Numbers, Booleans)

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

```
printId("abc123"); // Output: Your ID (string): ABC123
printId(123);      // Output: Your ID (number): 123.00
```

---

### ✓ b) instanceof Guard (Classes)

```
class Car {
  drive() {
    console.log("Driving a car");
  }
}

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

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

```
useVehicle(new Car()); // Output: Driving a car
useVehicle(new Bike()); // Output: Riding a bike
```

---

### ✓ c) in Operator Guard (Object Properties)

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

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

---

### ✅ d) Custom Type Guards (Advanced)

```
type Admin = { role: "admin"; name: string };
type User = { role: "user"; name: string };

function isAdmin(person: Admin | User): person is Admin {
  return person.role === "admin";
}

function showPanel(person: Admin | User) {
  if (isAdmin(person)) {
    console.log("Admin Panel Access:", person.name);
  } else {
    console.log("User Dashboard:", person.name);
  }
}
```

---

## 🎓 Summary Table

Concept	How It Works	Example Tool
Type Safety	Ensures correct types	<code>let age: number</code>
Type Inference	TS infers types automatically	<code>let name = "Kiran"</code>
Type Guards	Runtime checks to narrow types	<code>typeof</code> , <code>instanceof</code> , <code>in</code> , custom guards

---

## 🚩 Want a mini project idea to practice?

Create a **User Management System** where:

- User can be either `Admin`, `Member`, or `Guest`.
- Use **type guards** to show different dashboard options.
- Practice **inference** and **type safety** in form input validation.

Would you like me to provide that as a project template?

## Enums `enum` Declare a set of named constants. Example: `enum Direction { Up, Down }`

Absolutely, Routh! Let's dive deep into **Enums** in **TypeScript**, from beginner to advanced. I'll explain each level with **well-commented examples** so you can build strong problem-solving skills step-by-step.

---

### What is an Enum?

An **enum** (short for **enumeration**) is a way to define a set of **named constants** — making your code more readable and less error-prone.

---

### Level 1: Basic Enum

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

```
// Usage  
let move: Direction = Direction.Left;  
console.log(move); // Output: 2 (default starts from 0)
```

 By default, `Up = 0`, `Down = 1`, etc.

---

### Level 2: Custom Number Values

```
enum StatusCode {  
  OK = 200,  
  NotFound = 404,  
  ServerError = 500  
}
```

```
console.log(StatusCode.OK); // 200
console.log(StatusCode[404]); // NotFound (Reverse mapping)
```

✓ **Reverse Mapping** works with number enums.

---

## **Level 3: String Enums**

```
enum LogLevel {
  Info = "INFO",
  Warn = "WARN",
  Error = "ERROR"
}

function log(level: LogLevel, message: string) {
  console.log(`[${level}] ${message}`);
}

log(LogLevel.Warn, "Low disk space");
// Output: [WARN] Low disk space
```

⊘ No reverse mapping in **string enums**.

---

## **Level 4: Heterogeneous Enums (✗ Avoid if possible)**

```
enum Mixed {
  No = 0,
  Yes = "YES"
}
```

⚠ This mixes numbers and strings. It's allowed but discouraged for clarity.

---

## **Level 5: Enums in Functions (Real use case)**

```
enum TrafficLight {
  Red,
```

```
Yellow,  
Green  
}
```

```
function getAction(light: TrafficLight): string {  
  switch (light) {  
    case TrafficLight.Red:  
      return "Stop";  
    case TrafficLight.Yellow:  
      return "Slow down";  
    case TrafficLight.Green:  
      return "Go";  
  }  
}
```

```
console.log(getAction(TrafficLight.Yellow)); // "Slow down"
```

🧠 Improves maintainability and type safety.

---

## 🧱 Level 6: Enum as Flags (Bitwise operations)

```
enum Permission {  
  Read = 1 << 0, // 0001  
  Write = 1 << 1, // 0010  
  Execute = 1 << 2 // 0100  
}
```

```
let userPermission = Permission.Read | Permission.Write;
```

```
function hasPermission(perms: number, check: Permission): boolean {  
  return (perms & check) === check;  
}
```

```
console.log(hasPermission(userPermission, Permission.Write)); // true  
console.log(hasPermission(userPermission, Permission.Execute)); // false
```

👉 Great for **access control systems**.

---

## Level 7: Enum with Computed & Constant Members

```
const responseTime = () => 100;
```

```
enum Speed {  
  Slow = 10,  
  Medium = responseTime(), // computed member  
  Fast = 100  
}
```

⚠ Only **last** members can be **computed**.

---

## Level 8: Const Enums (For Performance)

```
const enum Direction {  
  Up,  
  Down,  
  Left,  
  Right  
}
```

```
let dir = Direction.Up;  
console.log(dir); // compiled as plain number (no object in JS)
```

✅ Eliminates enum object at runtime for performance.

---

## Bonus: Enum in Class

```
enum Category {  
  Food,  
  Electronics,  
  Clothing  
}
```

```
class Product {  
  constructor(public name: string, public category: Category) {}  
}
```

```
let laptop = new Product("Laptop", Category.Electronics);
```



```
console.log(laptop); // Product { name: 'Laptop', category: 1 }
```

---

## Summary Table

Enum Type	Reverse Mapping	Use Case
Numeric Enum	✓ Yes	Basic options, default values
String Enum	✗ No	Labels, logs, readability
Const Enum	✗ No	Performance-critical code
Bitwise Enum	✗ No	Permission, flag checks
Computed Enum	⚠ Partial	Rarely used, dynamic assignment

---

Would you like a small TypeScript project using enums for **real-world scenarios** like To-do apps, games, or authentication? I can build one for you.

## Advanced Types Union, Intersection, Literal, Tuple, Mapped, Conditional, Utility Types Complex type operations. Example: `type A = "yes"`

Absolutely, Routh Kiran Babu! Below is a comprehensive beginner-to-advanced **guide on Advanced Types in TypeScript**, packed with **clear explanations** and **powerful examples** so you master them step by step.

---

## Advanced Types in TypeScript — From Beginner to Advanced

---

### 1. Union Types (|)

A value can be **one of several types**.

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

function handleStatus(status: Status) {
  if (status === "success") console.log("Success!");
  else if (status === "error") console.log("Error!");
  else console.log("Loading...");
}
```

---

## 2. Intersection Types (&)

A value **must satisfy all** given types (combine multiple types).

```
type User = { name: string };
type Admin = { role: "admin" };

type AdminUser = User & Admin;

const admin: AdminUser = {
  name: "Routh",
  role: "admin"
};
```

---

## 3. Literal Types

You specify **exact values** a variable can take.

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

const move = (dir: Direction) => {
  console.log(`Moving ${dir}`);
};

move("left");
```

---

## 4. Tuple Types

An array with **fixed length and types** at specific positions.

```
type RGB = [number, number, number];
```

```
const red: RGB = [255, 0, 0];  
const green: RGB = [0, 255, 0];
```

You can also label tuple members (improves readability):

```
type Point = [x: number, y: number];  
  
const p: Point = [5, 10];
```

---

## 5. Mapped Types

Create new types by looping over keys of another type.

```
type User = {  
  name: string;  
  age: number;  
};  
  
type ReadonlyUser = {  
  readonly [K in keyof User]: User[K];  
};
```

// Equivalent to:

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

---

## 6. Conditional Types

Types that **depend on a condition**.

```
type IsString<T> = T extends string ? "Yes" : "No";  
  
type Test1 = IsString<string>; // "Yes"  
type Test2 = IsString<number>; // "No"
```

More complex:

```
type Flatten<T> = T extends Array<infer U> ? U : T;
```

```
type A = Flatten<string[]>; // string
type B = Flatten<number>; // number
```

---

## 7. Utility Types (Built-in helpers)

Very useful for transforming types.

Utility	Description
<code>Partial&lt;T&gt;</code>	All properties optional
<code>Required&lt;T&gt;</code>	All properties required
<code>Readonly&lt;T&gt;</code>	All properties readonly
<code>Pick&lt;T, K&gt;</code>	Pick specific keys
<code>Omit&lt;T, K&gt;</code>	Omit specific keys
<code>Record&lt;K, T&gt;</code>	Create object with keys K and values T

### Examples:

```
type Todo = {
  title: string;
  completed: boolean;
};
```

```
type OptionalTodo = Partial<Todo>;
```

```
type RequiredTodo = Required<OptionalTodo>;
```

```
type ReadonlyTodo = Readonly<Todo>;
```

```
type TodoTitle = Pick<Todo, "title">;
```

```
type TodoWithoutCompleted = Omit<Todo, "completed">;
```

```
type StatusMap = Record<"success" | "fail", number>;
```

```
const status: StatusMap = {  
  success: 200,  
  fail: 500,  
};
```

---

### Combined Example (Complex Type Operation)

```
type ResponseType = "success" | "error";
```

```
type ApiResponse<T> = {  
  status: ResponseType;  
  data: T extends "success" ? { result: string } : { error: string };  
};
```

```
const response1: ApiResponse<"success"> = {  
  status: "success",  
  data: { result: "Data fetched successfully" }  
};
```

```
const response2: ApiResponse<"error"> = {  
  status: "error",  
  data: { error: "Something went wrong" }  
};
```

---

### Summary Chart

Feature	Example Syntax	Use Case
Union	<code>`type A = "yes"`</code>	"no"
Intersection	<code>type A = B &amp; C</code>	Combine multiple types
Literal	<code>"open"</code>	Restrict to exact value
Tuple	<code>[number, string]</code>	Fixed-size array
Mapped	<code>[K in keyof T]: T[K]</code>	Loop over keys

Conditional `T extends U ? X : Y`

Type depends on another


Utility Types `Partial<T>`, `Omit<T, K>`,  
`Pick<T, K>`

Type transformations

---

Would you like a PDF or TypeScript file with all these examples? Or next-level examples like:

- Discriminated unions
  - Template literal types
  - Infer keyword use in conditionals
- Let me know and I'll build the next part for you!


 **Modules & Namespaces namespace (deprecated)** TypeScript supports namespace and more structured module systems.

Absolutely! Let's walk through **Modules** and **Namespaces** in **TypeScript**—focusing on **best practices**, **real-world examples**, and **how to transition from beginner to advanced**.

---

## Modules vs Namespaces in TypeScript

### Summary

Feature	Namespaces	Modules (Recommended  )
Usage	<code>namespace MyNamespace {}</code>	<code>export, import</code>
Scope	Global (via <code>&lt;script&gt;</code> or <code>reference</code> )	File-based
Compilation	Needs <code>--outFile</code>	CommonJS, ES Modules, etc.

Modern  
Usage

✗ Deprecated for modular apps

✓ Preferred for all modern projects

Best For

Small scripts

Large apps & team development

---

## 1. Namespaces (Deprecated for large-scale apps)

! Use only for backward compatibility or simple HTML projects.

### ✓ Basic Namespace Example

```
// file: shape.ts
namespace Shapes {
  export class Circle {
    constructor(public radius: number) {}

    area(): number {
      return Math.PI * this.radius ** 2;
    }
  }
}
```

```
// file: main.ts
/// <reference path="shape.ts" />
const c = new Shapes.Circle(5);
console.log("Area:", c.area());
```

### ⚠ Compile With:

```
tsc --outFile app.js shape.ts main.ts
```

🧠 Use `/// <reference path="..." />` to link files in the right order.

---

## ✓ 2. Modules (Recommended)

Works with modern JavaScript systems like **CommonJS**, **ESModules**, and **Webpack/Vite**.

#### **mathUtils.ts**

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

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

#### **app.ts**

```
import { add, multiply } from './mathUtils';
```

```
console.log("Sum:", add(10, 5));  
console.log("Product:", multiply(10, 5));
```

### **Compile:**

tsc

Make sure **tsconfig.json** has **module**: "ES6" or "CommonJS"

---

## **Module Variations: Intermediate to Advanced**

### **1 Default Export**

```
// logger.ts  
export default function log(message: string) {  
  console.log("LOG:", message);  
}
```

```
// main.ts  
import log from './logger';
```



```
log("App started");
```

## 2 Re-exporting

```
// file: utils.ts
export * from './mathUtils';
export * from './stringUtils';
```

```
// main.ts
import { add } from './utils';
```

## 3 Dynamic Imports (Advanced)

```
async function loadMath() {
  const math = await import('./mathUtils');
  console.log(math.add(1, 2));
}
```

## 4 Import with Alias

```
import * as MathUtils from './mathUtils';
console.log(MathUtils.add(3, 4));
```

---



# Real-World Module Structure (Advanced)

```
src/
├── utils/
│   ├── mathUtils.ts
│   └── stringUtils.ts
├── services/
│   └── apiService.ts
├── models/
│   └── user.ts
```

|  
└─ index.ts

### mathUtils.ts

```
export const PI = 3.14;  
  
export function circleArea(radius: number) {  
  return PI * radius ** 2;  
}
```

### index.ts

```
import { circleArea } from './utils/mathUtils';  
  
console.log("Area:", circleArea(10));
```

---

## Tips for Beginners → Advanced

Level	What to Learn	Practice Ideas
Beginner	<code>export</code> , <code>import</code> basics	Math utilities
Intermediate	<code>default</code> vs named exports, aliasing	Logger, API service
Advanced	Dynamic import, re-exporting, bundling	Modular app architecture

---

## When to Use What?

Scenario	Use
Web app with bundler (Vite/Webpack)	<input checked="" type="checkbox"/> Modules
Node.js backend	<input checked="" type="checkbox"/> Modules

Tiny HTML + TypeScript experiment	Namespaces (optional)
Modern production app	✓ Modules

---

## ✓ Conclusion

✓ **Always use modules** for maintainability, scalability, and team collaboration.

Namespaces are only useful when **modules aren't available** (e.g., basic script tags in HTML). Modern TypeScript development should always favor **ES Modules**.

---

Would you like me to generate a full **project structure with modules** or create **interactive module** 🏗️ **Decorators** `@decorator` syntax  
Experimental feature to annotate and modify classes and properties. JavaScript has only recently added experimental support. examples using TypeScript Playground?

Absolutely, Routh Kiran Babu! Let's explore **Decorators** in **TypeScript**—from beginner to advanced level—with simple and gradually advancing **examples**.

---

### 🔥 Before You Begin

To use decorators, you **must enable experimental support** in your `tsconfig.json`:

```
{
  "experimentalDecorators": true,
  "target": "ES6"
}
```

---

# What Are Decorators?

Decorators are special functions prefixed with `@` that can annotate and modify:

- Classes
- Methods
- Properties
- Accessors
- Parameters

They are **higher-order functions** that can intercept and extend behavior.

---


## 1. Basic Class Decorator

```
function Logger(constructor: Function) {  
  console.log(`Class created: ${constructor.name}`);  
}
```

```
@Logger  
class User {  
  constructor() {  
    console.log("User instance created.");  
  }  
}
```

 Output:

```
Class created: User  
User instance created.
```

 **Use case:** Logging when classes are defined.

---

## 2. Property Decorator

```
function Capitalize(target: any, propertyKey: string) {
  let value = "";

  const getter = () => value;
  const setter = (newVal: string) => {
    value = newVal.toUpperCase();
  };

  Object.defineProperty(target, propertyKey, {
    get: getter,
    set: setter,
  });
}

class Person {
  @Capitalize
  name: string = "";
}

const p = new Person();
p.name = "john";
console.log(p.name); // Output: JOHN
```

✅ **Use case:** Format or validate data automatically.

---

### 3. Method Decorator

```
function LogMethod(
  target: any,
  propertyName: string,
  descriptor: PropertyDescriptor
) {
  const originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${propertyName} with args:`, args);
    return originalMethod.apply(this, args);
  };
}

class Calculator {
  @LogMethod
```

```

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

const calc = new Calculator();
calc.add(2, 3); // Logs: Calling add with args: [2, 3]

```

✅ **Use case:** Logging or tracking method calls.

---

## 4. Accessor Decorator

```

function CheckPositive(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  const originalSet = descriptor.set!;
  descriptor.set = function (value: number) {
    if (value < 0) throw new Error("Value must be positive");
    originalSet.call(this, value);
  };
}

```

```

class BankAccount {
  private _balance = 0;

  @CheckPositive
  set balance(val: number) {
    this._balance = val;
  }

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

```

```

const acc = new BankAccount();
acc.balance = 100; // OK
// acc.balance = -50; // Error: Value must be positive

```

✓ **Use case:** Validating input to setters.

---

## 5. Parameter Decorator

```
function LogParameter(target: Object, methodName: string, index: number) {  
  console.log(`Parameter index ${index} of method ${methodName} was decorated.`);  
}
```

```
class Service {  
  greet(@LogParameter message: string) {  
    console.log(message);  
  }  
}
```

✓ **Use case:** Metadata collection for dependency injection.

---



## 6. Decorator Factory (with parameters)

```
function Role(role: string) {  
  return function (constructor: Function) {  
    constructor.prototype.role = role;  
  };  
}
```

```
@Role("admin")  
class AdminPanel {}
```

```
const admin = new AdminPanel();  
console.log((admin as any).role); // admin
```

✓ **Use case:** Customize decorators with arguments.

---



## 7. Compose Multiple Decorators

```
function First() {  
  return function (target: Function) {  
    console.log("First decorator");  
  };  
}
```

```

    };
}

function Second() {
  return function (target: Function) {
    console.log("Second decorator");
  };
}

@First()
@Second()
class Test {}

```

 Output:

```

Second decorator
First decorator

```

 **Note:** They execute **bottom-up**.

## 8. Real-Life Use Case: Authentication

```


function Authenticated(target: any, methodName: string, descriptor: PropertyDescriptor) {
  const original = descriptor.value;
  descriptor.value = function (...args: any[]) {
    if (!this.isLoggedIn) {
      throw new Error("Not authenticated!");
    }
    return original.apply(this, args);
  };
}

class Dashboard {
  isLoggedIn = false;

  @Authenticated
  viewSecretData() {
    console.log("Secret Data Viewed");
  }
}

```



```
const dash = new Dashboard();
// dash.viewSecretData(); // Error: Not authenticated!
dash.isLoggedIn = true;
dash.viewSecretData(); //  Secret Data Viewed
```


---

## Summary Table

Decorator Type	Target	Syntax	Use Case
Class	Class	<code>@Logger</code>	Logging, role tagging
Property	Class prop	<code>@Capitalize</code>	Auto-formatting
Method	Function	<code>@LogMethod</code>	Logging, access control
Accessor	Getter/setter	<code>@CheckPositive</code>	Validation
Parameter	Function arg	<code>@LogParameter</code>	Metadata, injection
Factory	Any	<code>@Decorator(args)</code>	Customizable decorator
Multiple Decorators	Any	<code>@First()\n@Second()</code>	Composition

---

Would you like a **hands-on mini-project** using decorators like role-based access, input validation, and logging in one place?

 **Abstract Classes**    `abstract class`    Define base classes with abstract methods. JavaScript does not have built-in support for abstract methods.

Absolutely, Routh Kiran Babu! Let's go step-by-step to understand **Abstract Classes** in **TypeScript**, starting from beginner level and building up to more advanced concepts.

---

# Abstract Classes in TypeScript

## What is an Abstract Class?

- An **abstract class** is a class that **cannot be instantiated directly**.
  - It can contain **abstract methods** (without implementation) and **concrete methods** (with implementation).
  - **Purpose:** To define a **blueprint** for other classes.
- 

## Beginner Level: Basic Abstract Class Example

### Example: **Animal** abstract class

```
abstract class Animal {  
  abstract makeSound(): void;  
  
  move(): void {  
    console.log("The animal moves.");  
  }  
}  
  
class Dog extends Animal {  
  makeSound(): void {  
    console.log("Woof! Woof!");  
  }  
}  
  
const myDog = new Dog();  
myDog.makeSound(); // Woof! Woof!  
myDog.move();      // The animal moves.
```

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

### Explanation:

- **Animal** is an abstract class.

- `makeSound()` is an **abstract method**—must be implemented by subclasses.
  - `Dog` extends `Animal` and provides implementation for `makeSound()`.
- 

## Intermediate Level: Abstract Properties and Multiple Subclasses

### Example: Shape Calculation

```
abstract class Shape {  
    abstract area(): number;  
    abstract perimeter(): number;  
}
```

```
class Circle extends Shape {  
    constructor(private radius: number) {  
        super();  
    }
```

```
    area(): number {  
        return Math.PI * this.radius ** 2;  
    }
```

```
    perimeter(): number {  
        return 2 * Math.PI * this.radius;  
    }  
}
```

```
class Rectangle extends Shape {  
    constructor(private width: number, private height: number) {  
        super();  
    }
```

```
    area(): number {  
        return this.width * this.height;  
    }
```

```
    perimeter(): number {  
        return 2 * (this.width + this.height);  
    }
```

```
}
```

```
// Usage
```

```
const circle = new Circle(5);  
console.log("Circle Area:", circle.area());    // Circle Area: 78.5398  
console.log("Circle Perimeter:", circle.perimeter());
```

```
const rect = new Rectangle(10, 20);  
console.log("Rectangle Area:", rect.area());    // Rectangle Area: 200  
console.log("Rectangle Perimeter:", rect.perimeter());
```

---



## Advanced Level: Abstract Class with Generics and Dependency Injection



### Example: Generic Repository Pattern (like in backend apps)

```
interface Entity {  
  id: number;  
}  
  
abstract class Repository<T extends Entity> {  
  abstract getById(id: number): T | undefined;  
  abstract save(entity: T): void;  
}  
  
class User implements Entity {  
  constructor(public id: number, public name: string) {}  
}  
  
class UserRepository extends Repository<User> {  
  private users: User[] = [];  
  
  getById(id: number): User | undefined {  
    return this.users.find(user => user.id === id);  
  }  
  
  save(user: User): void {  
    this.users.push(user);  
  }  
}
```

```
// Usage
const userRepo = new UserRepository();
userRepo.save(new User(1, "Kiran"));
console.log(userRepo.getById(1)); // User { id: 1, name: 'Kiran' }
```

### Concepts used:

- **Generics** for reusable type-safe repositories.
- **Abstract classes** as a blueprint for data access logic.

---

## ! Key Rules and Tips

Rule/Tip	Explanation
<code>abstract</code> keyword	Used to define abstract class or methods.
No instantiation of abstract class	<code>new AbstractClass()</code> will throw an error.
Must extend and implement	Subclass must implement all abstract methods.
Abstract class $\neq$ Interface	Abstract class can have concrete logic. Interface can't.

---

## Real-World Use Cases

Use Case	Example
UI Components	<pre>abstract class UIComponent {   render(): void }</pre>
Database Repositories	<pre>abstract class Repository&lt;T&gt; {   getAll(): T[] }</pre>
Game Engines	<pre>abstract class GameObject { update(): void }</pre>

---

Would you like a **TypeScript playground link**, **project-based mini app**, or **interview-level abstract class challenges** next?



## Compile-Time Checking Compile-Time Errors

TypeScript can catch many bugs at compile time, unlike JavaScript which catches at runtime.

Absolutely! Let's walk through the concept of **Compile-Time Checking** in TypeScript with clear beginner-to-advanced examples. This is one of TypeScript's **biggest advantages** over JavaScript — catching errors **before the code even runs**.

---



## What is Compile-Time Checking?

TypeScript **analyzes your code** when you compile (`tsc`), catching common mistakes like:

- Using the wrong type
- Accessing properties that don't exist
- Passing incorrect parameters to functions



### JavaScript: Errors only appear at runtime

```
function greet(name) {  
  console.log("Hello, " + name.toUpperCase());  
}
```

`greet(42);` // ❌ Runtime Error: name.toUpperCase is not a function

You won't see any issue until the code runs. 🚨

---



### TypeScript: Errors appear at compile time

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

```
greet(42);  
// ❌ Compile-Time Error: Argument of type 'number' is not assignable to parameter of type  
'string'.
```

Now let's go from beginner to advanced with **compile-time checking examples**.

---

## Beginner Examples

### 1. Basic Type Mismatch

```
let age: number = "twenty";  
// ❌ Error: Type 'string' is not assignable to type 'number'.
```

---

### 2. Function Parameter Type Checking

```
function multiply(a: number, b: number): number {  
  return a * b;  
}  
  
multiply(5, "2");  
// ❌ Error: Argument of type 'string' is not assignable to parameter of type 'number'.
```

---

### 3. Accessing Non-Existent Properties

```
let user = {  
  name: "Ravi",  
  age: 25  
};  
  
console.log(user.email);  
// ❌ Error: Property 'email' does not exist on type '{ name: string; age: number; }'
```

---

## Intermediate Examples

### 4. Wrong Return Type

```
function getUser(): string {  
  return { name: "Ravi" };  
  // ❌ Error: Type '{ name: string; }' is not assignable to type 'string'.  
}
```

---

## 5. Interface Mismatch

```
interface Product {  
  name: string;  
  price: number;  
}  
  
const item: Product = {  
  name: "Laptop",  
  price: "30000"  
  // ❌ Error: Type 'string' is not assignable to type 'number'.  
};
```

---

## 6. Missing Properties

```
interface Car {  
  model: string;  
  year: number;  
}  
  
const car: Car = {  
  model: "Tesla"  
  // ❌ Error: Property 'year' is missing in type.  
};
```

---

## Advanced Examples

### 7. Incorrect Generic Type

```
function printArray<T>(arr: T[]) {  
  arr.forEach(item => console.log(item));  
}  
  
printArray<number>(["a", "b", "c"]);
```



// ❌ Error: Type 'string' is not assignable to type 'number'.

---

## 8. Invalid Union Type Usage

```
type ID = number | string;
```

```
function printId(id: ID) {  
  console.log(id.toFixed(2));  
  // ❌ Error: Property 'toFixed' does not exist on type 'string | number'.  
}
```

✅ Fix:

```
if (typeof id === "number") {  
  console.log(id.toFixed(2));  
}
```

---

## 9. Incorrect Enum Usage

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

```
function move(dir: Direction) {  
  console.log("Moving", dir);  
}
```

```
move("Left");
```

// ❌ Error: Argument of type '"Left"' is not assignable to parameter of type 'Direction'

✅ Fix:

```
move(Direction.Left);
```

---

## 10. Wrong Type in Class Property

```

class Student {
  name: string;
  age: number;

  constructor(name: string, age: string) {
    this.name = name;
    this.age = age;
    // ❌ Error: Type 'string' is not assignable to type 'number'.
  }
}

```

---



## Bonus Tip: Enable Strict Mode in **tsconfig.json**

To catch **maximum compile-time errors**, set this:

```

{
  "compilerOptions": {
    "strict": true
  }
}

```

---



## Summary Table

JavaScript	TypeScript (Compile-Time Error)
<code>greet(42)</code>	Argument of type 'number' is not assignable to 'string'
<code>user.email</code>	Property 'email' does not exist on type
<code>return {name: "Ravi"};</code>	Type '{ name: string; }' is not assignable to type 'string'
<code>price: "3000" in Product</code>	Type 'string' is not assignable to type 'number'

---

Would you like me to provide **interactive exercises or quizzes** to test your understanding of compile-time checks?



## Declaration Files.d.ts Files Define types for existing JavaScript libraries, enabling IntelliSense and type checking.

Absolutely! Here's a beginner-to-advanced guide on **Declaration Files** (`.d.ts` files) in **TypeScript**, with practical examples to help you fully understand how to define types for existing JavaScript libraries, enabling IntelliSense and type checking.

---



### What Are Declaration Files (`.d.ts`)?

- Declaration files in TypeScript **only contain type information** — no actual JavaScript code.
  - Useful when using **plain JavaScript libraries** in a TypeScript project to enable:
    - **IntelliSense**
    - **Compile-time type checking**
- 



### When Do You Need `.d.ts` Files?

1. You're using a **JavaScript library** that doesn't ship with TypeScript support.
  2. You're writing a **JavaScript library** and want to provide types to consumers.
  3. You want to extend or modify existing types.
- 



## Beginner Example: Declaring a Simple JavaScript Function

## 1. JavaScript Library (**math.js**)

```
// math.js
function add(a, b) {
  return a + b;
}
```

## 2. Create a Declaration File: **math.d.ts**

```
// math.d.ts
declare function add(a: number, b: number): number;
```

Place both **math.js** and **math.d.ts** in the same folder or configure **typeRoots** in **tsconfig.json**.

## 3. TypeScript File Using It

```
// index.ts
/// <reference path="./math.d.ts" />

console.log(add(5, 3)); // Works with IntelliSense and type checking
```

---

# Intermediate Example: Declaring a Module

Let's say you're using a JavaScript file as a module (e.g., with **import/export**).

## 1. **logger.js** (No types)

```
// logger.js
export function log(message) {
  console.log(message);
}
```

## 2. **logger.d.ts**

```
// logger.d.ts
declare module "logger" {
```

```
export function log(message: string): void;
}
```

### 3. Usage in TypeScript

```
// app.ts
import { log } from "logger";

log("Hello TypeScript!"); // Typed!
```

---



## Intermediate: Declaring Object Types in .d.ts

### 1. config.js

```
module.exports = {
  apiKey: "abcd-1234",
  timeout: 5000,
};
```

### 2. config.d.ts

```
declare module "config" {
  export const apiKey: string;
  export const timeout: number;
}
```

### 3. Use in TypeScript

```
import { apiKey, timeout } from "config";
console.log(apiKey, timeout);
```

---

# Advanced Example: Declaring Classes, Interfaces, Overloads

## 1. `utils.js`

```
function greet(name) {  
    return "Hello, " + name;  
}  
  
function greet(name, age) {  
    return `Hello, ${name}. You are ${age} years old.`;  
}
```

## 2. `utils.d.ts`

```
declare function greet(name: string): string;  
declare function greet(name: string, age: number): string;
```

---

## Declaring a Class

```
// library.d.ts  
declare class Person {  
    constructor(name: string, age: number);  
    greet(): string;  
    readonly age: number;  
}
```

## Usage

```
const p = new Person("Ravi", 30);  
console.log(p.greet());
```

---

# Advanced: Declaring Global Variables and Namespaces

## Global Variable Example

```
// globals.d.ts
declare var VERSION: string;

console.log(VERSION); // Type-checked as string
```

---

## Namespace Example

```
// shapes.d.ts
declare namespace Shapes {
  interface Circle {
    radius: number;
  }

  interface Rectangle {
    width: number;
    height: number;
  }

  function draw(shape: Circle | Rectangle): void;
}
```

## Usage

```
Shapes.draw({ radius: 10 });
Shapes.draw({ width: 10, height: 20 });
```

---



## Pro Tip: Use DefinitelyTyped (@types)

You often don't have to write `.d.ts` files from scratch!

```
npm install --save-dev @types/lodash
```

Then in your TypeScript:

```
import _ from 'lodash';
```

```
_.chunk(['a', 'b', 'c', 'd'], 2); // Full IntelliSense
```

---




## Summary Table

Use Case	Syntax Example
Declare function	<code>declare function fn(a: number): void;</code>
Declare module	<code>declare module 'lib' { export function ... }</code>
Declare global var	<code>declare var VERSION: string;</code>
Declare class	<code>declare class User { constructor(...); method(): void; }</code>
Declare interface	<code>declare interface Car { speed: number; }</code>
Declare namespace	<code>declare namespace MyLib { ... }</code>

---

Would you like me to generate a GitHub-ready folder or project for these examples with proper `tsconfig.json`, `.js`, and `.d.ts` files?

 **Readonly and Access Modifiers** `private`, `protected`, `readonly`, `public` Enforce access restrictions on class members. JavaScript now has `#private`, but TS has stronger compile-time enforcement.

Absolutely, Routh! Let's walk through **Readonly and Access Modifiers** in **TypeScript**, with **easy-to-understand examples** that gradually go from **beginner** to **advanced** level. These modifiers help **encapsulate** and **protect your data**.

---

### 1. Basic Concept





Modifier	Description
<code>public</code>	Accessible anywhere (default in TS).
<code>private</code>	Accessible <b>only within the class</b> .
<code>protected</code>	Accessible <b>within the class and subclasses</b> .
<code>readonly</code>	Can be <b>read but not changed</b> after initialization.

---

## Beginner Examples

### 1. `public` (Default Access)

```
class Student {  
  public name: string;  
  
  constructor(name: string) {  
    this.name = name;  
  }  
  
  greet() {  
    console.log(`Hello, I'm ${this.name}`);  
  }  
}  
  
const s1 = new Student("Ravi");  
s1.greet();      //  Accessible  
console.log(s1.name); //  Accessible
```

---

### 2. `private`

```
class BankAccount {  
  private balance: number;  
  
  constructor(initialBalance: number) {  
    this.balance = initialBalance;  
  }  
}
```

```
showBalance() {  
  console.log(`Balance: ₹${this.balance}`);  
}  
}  
  
const acc = new BankAccount(1000);  
acc.showBalance(); // ✅ Allowed  
// console.log(acc.balance); ❌ Error: Property 'balance' is private
```

---

### 🟡 3. **protected**

```
class Animal {  
  protected species: string;  
  
  constructor(species: string) {  
    this.species = species;  
  }  
}  
  
class Dog extends Animal {  
  bark() {  
    console.log(`I am a ${this.species} and I bark.`);  
  }  
}  
  
const dog = new Dog("Canine");  
dog.bark(); // ✅ Allowed  
// console.log(dog.species); ❌ Error: 'species' is protected
```

---

### 🟢 4. **readonly**

```
class Vehicle {  
  readonly wheels: number = 4;  
  
  constructor() {  
    // this.wheels = 6; ❌ Error: Cannot assign to 'wheels'  
  }  
  
  info() {  
    console.log(`Vehicle has ${this.wheels} wheels`);  
  }  
}
```


```
}
```

```
const car = new Vehicle();  
car.info();
```

---

## Intermediate Examples

### readonly with Constructor Initialization




```
class User {  
  readonly id: number;  
  name: string;  
  
  constructor(id: number, name: string) {  
    this.id = id;    //  OK to assign once in constructor  
    this.name = name;  
  }  
}
```

```
const user = new User(1, "Kiran");  
// user.id = 2;  Error
```

---

### Access Modifiers in Constructor Parameters (Shortcut)

```
class Point {  
  constructor(  
    public x: number,  
    private y: number,  
    protected z: number  
  ) {}  
  
  getY() {  
    return this.y;  
  }  
}
```

```
const p = new Point(1, 2, 3);  
console.log(p.x);    //  public  
// console.log(p.y);  private  
// console.log(p.z);  protected
```

---

## Advanced Example

### ✓ Combine All: **private**, **protected**, **public**, and **readonly**

```
class Employee {
  public name: string;
  private salary: number;
  protected role: string;
  readonly empld: string;

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

  getDetails() {
    return `${this.name} has ID ${this.empld} and role ${this.role}`;
  }

  getSalary() {
    return `Salary is ₹${this.salary}`;
  }
}

class Manager extends Employee {
  promote() {
    console.log(`${this.name} is promoted to Senior ${this.role}`);
  }
}

const e1 = new Manager("Anil", 60000, "Manager", "EMP001");
console.log(e1.getDetails()); // ✓ public
console.log(e1.getSalary()); // ✓ private accessed internally
e1.promote(); // ✓ protected accessed from subclass
// e1.salary = 80000; // ✗ Error: private
// e1.empld = "EMP999"; // ✗ Error: readonly
```

---

## Bonus: JavaScript's **#private** Fields vs TypeScript **private**

JavaScript supports **#private** (hard privacy), while TypeScript's **private** is at compile-time.

```
class JSStylePrivate {
  #secret = "hidden"; // JavaScript private field

  getSecret() {
    return this.#secret;
  }
}

const obj = new JSStylePrivate();
// console.log(obj.#secret); ❌ Syntax Error
console.log(obj.getSecret()); // ✅ Output: hidden
```

---

### ✅ Summary Table

Modifier	Scope	Use Case
<b>public</b>	Accessible anywhere	General access
<b>private</b>	Class only	Hide sensitive data
<b>protected</b>	Class + Subclasses	Share with subclasses only
<b>readonly</b>	Assigned once, can be read	Immutable property (e.g. ID)
<b>#private</b>	JS native private, accessed by <b>#field</b>	Hard privacy (new JS syntax)

---

Would you like **TypeScript practice exercises**, or want these examples in a downloadable **PDF** or **code repo** format?



# Optional Chaining + Nullish Coalescing with Types

Strong type checks with `?.` and `??`

JavaScript has these operators, but TypeScript adds type safety with them.

Absolutely! You're diving into a very **important and powerful** feature in TypeScript: **Optional Chaining (`?.`)** and **Nullish Coalescing (`??`)**, which provide **safe access to nested properties** and **defaulting only when values are null or undefined** — with **strong type checks** in TypeScript. Let's go step by step with clean and layered examples.

---

## ✓ 1. Optional Chaining (`?.`) – Type-Safe Access

### 🧠 What It Does:

Safely accesses nested object properties or methods **only if the object is not `null` or `undefined`**, otherwise returns `undefined`.

### 📖 Example: Safe Property Access

```
type User = {  
  name: string;  
  address?: {  
    city?: string;  
  };  
};
```

```
const user1: User = { name: "Alice", address: { city: "Delhi" } };  
const user2: User = { name: "Bob" }; // address is undefined
```

```
// ❌ Without optional chaining: Can throw error  
// console.log(user2.address.city); // ❌ Runtime error
```

```
// ✅ With optional chaining  
console.log(user1.address?.city); // "Delhi"  
console.log(user2.address?.city); // undefined
```

---

### Example: Safe Method Call

```
type Logger = {  
  log?: () => void;  
};
```

```
const logger1: Logger = {  
  log: () => console.log("Logging..."),  
};
```

```
const logger2: Logger = {}; // log is undefined
```

```
logger1.log?.(); //  Prints "Logging..."  
logger2.log?.(); //  Does nothing, no error
```

---

## 2. Nullish Coalescing (??) – Default When Null/Undefined


### What It Does:

Provides a **default value** only if the expression is **null** or **undefined** (not for **false**, **0**, **' '**, etc.).

### Example: Default Value for Missing Input

```
function greet(name?: string) {  
  const displayName = name ?? "Guest";  
  console.log(`Hello, ${displayName}`);  
}
```

```
greet("Kiran"); // Hello, Kiran  
greet();       // Hello, Guest
```

 `||` would treat empty strings (`' '`) and `0` as falsy — **??** is **safer** for such defaults.

---

## 3. Combined Usage: **?.** + **??** Together

## Practical Example: Deep Access with Fallback

```
type Config = {  
  theme?: {  
    darkMode?: boolean;  
  };  
};  
  
const config1: Config = { theme: { darkMode: true } };  
const config2: Config = {};  
  
const darkMode1 = config1.theme?.darkMode ?? false;  
const darkMode2 = config2.theme?.darkMode ?? false;  
  
console.log(darkMode1); // true  
console.log(darkMode2); // false
```

---

## 4. Bonus: Optional Chaining with Arrays and Functions

### Optional Element Access

```
const names = ["Alice", "Bob"];  
  
console.log(names?.[0]); // "Alice"  
console.log(names?.[10]); // undefined
```

### Optional Function Invocation

```
type Callback = {  
  onDone?: () => void;  
};  
  
const cb: Callback = {};  
cb.onDone?.(); // Does nothing safely
```

---

## TypeScript-Specific Type Safety

In JavaScript, `?.` and `??` work — but TypeScript ensures:




- You only use `?.` on types that may actually be optional or undefined.
- You get compile-time errors if you misuse it, e.g.:

```
const user = { name: "Kiran" };  
// ❌ Error: Property 'address' does not exist on type '{ name: string; }'  
console.log(user.address?.city);
```

TypeScript forces you to **explicitly type objects**, and warns you when **optional chaining** makes no sense.




---

## Summary Table

Feature	Syntax	Purpose
Optional Chaining	<code>obj?.prop</code>	Safe access of possibly null/undefined
Nullish Coalescing	<code>val ?? def</code>	Fallback <b>only if val is null/undefined</b>
Combined Usage	<code>obj?.val ?? def</code>	Full safety chain for deeply nested props
Type Safety Advantage		Ensures usage only on nullable types

---

## Next Level Practice Ideas

-  Use optional chaining in API responses (`response.data?.user?.name`)
-  Use nullish coalescing in form defaults (`formData.age ?? 18`)
-  Write a utility that safely reads a config from nested structure

---

Would you like some mini projects or code challenges that use these concepts together (e.g., form parser, API response formatter, config loader)?



## Type Alias `type SomeType = ...` Create shorthand names for complex types.

Absolutely, Routh! Let's go from **beginner to advanced** in understanding **Type Aliases** in **TypeScript**, with **clear examples**, step-by-step. This concept helps simplify complex types, reuse them, and improve code readability.

---



## Type Aliases in TypeScript

```
type SomeType = ...
```

**Purpose:** Create a shorthand name for a complex or custom type.

---



### 1. Beginner Level



#### Basic Syntax:

```
type Age = number;  
let myAge: Age = 25;
```




Age is an alias for `number`.

---



#### Example 1: Simple Object

```
type Person = {  
  name: string;  
  age: number;  
};  
  
const p1: Person = {  
  name: "Alice",  
  age: 30,  
};
```


 Now instead of repeating `{ name: string; age: number }`, just use `Person`.

---

### **Example 2: Union Types**

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

```
let currentStatus: Status = "success";
```

 Better than repeating string literals everywhere.

---

## **2. Intermediate Level**

### **Example 3: Array of Custom Types**

```
type Product = {  
  id: number;  
  name: string;  
};
```

```
type ProductList = Product[];
```

```
const products: ProductList = [  
  { id: 1, name: "Pen" },  
  { id: 2, name: "Notebook" },  
];
```

---

### **Example 4: Function Type Alias**

```
type GreetFunction = (name: string) => string;
```

```
const greet: GreetFunction = (name) => `Hello, ${name}!`;
```

---

### **Example 5: Optional & Readonly Properties**

```
type User = {  
  readonly id: number;
```

```
name: string;
email?: string; // optional
};
```

```
const user1: User = { id: 101, name: "John" };
// user1.id = 102; ❌ Error: id is readonly
```

---

## 3. Advanced Level

### Example 6: Nested Type Aliases

```
type Address = {
  city: string;
  zip: string;
};

type Employee = {
  id: number;
  personal: {
    name: string;
    age: number;
  };
  address: Address;
};
```

---

### Example 7: Combining Aliases (Union + Intersection)

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

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

type UserRole = Admin | Guest;

const userA: UserRole = { role: "admin", accessLevel: 3 };
```

```
type WithTimestamp = {
  createdAt: Date;
  updatedAt: Date;
};

type Post = {
  title: string;
  content: string;
};

type TimestampedPost = Post & WithTimestamp;
```

---

### Example 8: Recursive Types (Advanced)

```
type JSONValue =
  | string
  | number
  | boolean
  | null
  | JSONValue[]
  | { [key: string]: JSONValue };

const data: JSONValue = {
  name: "Test",
  items: [1, 2, 3],
  details: {
    active: true,
    score: null,
  },
};
```

---

### When to Use **type** vs **interface**?

Feature	type	interface
Union Types	✓	✗
Tuples	✓	✗

Extension	✓ (with &)	✓ ( <code>extends</code> )
Declaration merging	✗	✓

Use `type` for unions/tuples. Use `interface` for extending object shapes (especially in class OOP patterns).

---

## Practice Challenge

- // 1. Create a type alias for a Rectangle with width and height
- // 2. Create a function that calculates area using the alias

```
type Rectangle = {
  width: number;
  height: number;
};
```

```
const calculateArea = (rect: Rectangle): number => rect.width * rect.height;
```

```
console.log(calculateArea({ width: 10, height: 5 })); // 50
```

---

Would you like a **project-based task** to apply all type alias concepts together (e.g., creating a Blog or E-commerce model using types)?

 **Structural Typing**      **Duck typing with**  
**structure-based compatibility**      **More rigidly**  
**enforced in TypeScript than in JavaScript.**

Absolutely! Let's walk through **Structural Typing** in **TypeScript** — a concept often described as "**Duck Typing**". We'll explore it with **simple to advanced examples**, so even beginners can understand how it works and why it's important.

---

## What is Structural Typing (Duck Typing)?

**Definition:** In TypeScript, **structural typing** means two types are compatible **if their structure matches**, regardless of their declared names.

✅ "If it walks like a duck and quacks like a duck, it's a duck." 🦆

---

## Real-World Analogy

Imagine two people:

- John: { name: "John", age: 25 }
- Jane: { name: "Jane", age: 30 }

Even if they're from different places, as long as both have **name** and **age**, TypeScript considers them structurally the same.

---

## ✅ Beginner Example

```
// Define a structure/type
interface Person {
  name: string;
  age: number;
}

// Function that accepts Person
function greet(person: Person) {
  console.log(`Hello, ${person.name}!`);
}

// Object with matching structure
const user = { name: "Alice", age: 28 };


greet(user); // ✅ OK because structure matches
```

Even if **user** is not explicitly declared as **Person**, **it's still accepted** because it matches the shape.

---

## Mismatched Structure


```
const wrongUser = { name: "Bob" };
```

```
greet(wrongUser); //  Error: Property 'age' is missing
```


---


## Intermediate Example: Extra Properties Are Okay


```
const employee = { name: "Charlie", age: 32, salary: 50000 };
```

```
greet(employee); //  OK — extra properties are ignored as long as required structure is present
```

### But: TypeScript is stricter with object literals directly passed

```
greet({ name: "Daisy", age: 22, salary: 30000 }); //  Error in strict mode (Excess Property Check)
```

 To fix it, assign first:

```
const daisy = { name: "Daisy", age: 22, salary: 30000 };  
greet(daisy); //  OK
```

---

## Assigning Between Compatible Types

```
interface Animal {  
  name: string;  
}
```

```
class Dog {  
  name = "Puppy";  
}
```

```
let animal: Animal;  
let dog = new Dog();
```


```
animal = dog; //  OK — structure matches
```

---



## Advanced: Generic Structural Typing

```
function identity<T>(value: T): T {  
  return value;  
}
```

```
const userObj = { name: "Eve", age: 30 };  
const result = identity(userObj); // Structure is preserved  
console.log(result.name); //  Eve
```

---


## Comparing With Nominal Typing (Used in Other Languages)

In **Java** or **C#**, two classes/interfaces need to have **explicit relationship** (inheritance, etc.).


In **TypeScript**, **only structure matters**:

```
interface Point2D {  
  x: number;  
  y: number;  
}
```

```
interface Pixel {  
  x: number;  
  y: number;  
}
```

```
let point: Point2D = { x: 10, y: 20 };  
let pixel: Pixel = point; //  OK: same structure
```

```
function logCoordinates(p: Point2D) {  
  console.log(`(${p.x}, ${p.y})`);  
}
```

```
logCoordinates(pixel); //  OK
```

---

## Use Case: Safer APIs

```
interface ApiResponse {
```

```
    success: boolean;
    data: any;
}

function handleResponse(res: ApiResponse) {
    if (res.success) {
        console.log("Data received:", res.data);
    }
}

const serverResponse = {
    success: true,
    data: { id: 1, name: "Item" },
    statusCode: 200
};

handleResponse(serverResponse); // ✅ OK
```

---



## Best Practices

- ✅ Use interfaces to **describe shapes** of objects
  - ✅ Don't over-rely on class-based inheritance — **prefer shape-based typing**
  - ✅ Use **type aliases** or **interfaces** to model domain-specific data
  - ✅ Leverage structural typing to write more **flexible and reusable** functions
- 






## Summary

Feature	Description
<b>Structural Typing</b>	Type compatibility based on structure
<b>Duck Typing</b>	"If it looks like a duck..." principle
<b>Extra Properties</b>	Ignored if passed via variable, flagged in literals
<b>Safer APIs</b>	Helps ensure expected data format
<b>Common in TS</b>	Unlike Java/C#, structural typing is native

---

Would you like me to give:

-  Practice problems?
-  A playground link to try?
-  Structural vs Nominal typing comparison PDF?

Let me know how you'd like to go further!

 **Discriminated Unions**      Tagged unions to  
handle different shapes in type-safe ways      Not  
natively supported in JS.

Absolutely! Discriminated Unions (also called Tagged Unions or Algebraic Data Types) are a powerful **TypeScript-only** feature for safely modeling data that can take different shapes.

---

## What Are Discriminated Unions?

Discriminated Unions combine:

1. **Union types** — `A | B | C`
  2. A **common "discriminant" property** — often called `kind` or `type`
  3. **Type narrowing** via control flow (like `if` or `switch`)
- 

## Use Case Example (from beginner to advanced)

---

### Beginner Example: Modeling Shapes

```
type Circle = {  
  kind: "circle";  
  radius: number;
```

```

};

type Square = {
  kind: "square";
  sideLength: number;
};

type Shape = Circle | Square;

function getArea(shape: Shape): number {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius * shape.radius;
  } else if (shape.kind === "square") {
    return shape.sideLength * shape.sideLength;
  }
  // Exhaustiveness check
  const _exhaustive: never = shape;
  return _exhaustive;
}

const c: Shape = { kind: "circle", radius: 3 };
console.log(getArea(c)); // 28.27

```

### 🧩 Why it works:

- `kind` is the **discriminant**.
- TS narrows `shape`'s type based on `shape.kind`.

---

### ✅ Intermediate Example: Handling API Responses

```

type SuccessResponse = {
  status: "success";
  data: string[];
};

type ErrorResponse = {
  status: "error";
  message: string;
};

```

```
type APIResponse = SuccessResponse | ErrorResponse;
```

```
function handleResponse(response: APIResponse) {  
  switch (response.status) {  
    case "success":  
      console.log("Data:", response.data);  
      break;  
    case "error":  
      console.error("Error:", response.message);  
      break;  
  }  
}
```

### 🌱 Benefit:

You don't need to check for existence of properties like `data` or `message`. TypeScript **knows exactly** what to expect when `status === "error"`.

---

### ✅ Advanced Example: State Machine (like React useReducer)

```
type LoadingState = {  
  type: "loading";  
};
```

```
type LoadedState = {  
  type: "loaded";  
  payload: string;  
};
```

```
type ErrorState = {  
  type: "error";  
  error: string;  
};
```

```
type AppState = LoadingState | LoadedState | ErrorState;
```

```
function reducer(state: AppState, action: { type: string; payload?: any }): AppState {  
  switch (action.type) {  
    case "start":  
      return { type: "loading" };  
    case "success":  
      return { type: "loaded", payload: action.payload };  
    case "fail":
```

```
    return { type: "error", error: action.payload };
  default:
    return state;
}
}
```

---

## Best Practices

Rule	Description
✓ Use a <b>kind</b> or <b>type</b> field	Helps TypeScript narrow types
✓ Add <b>never</b> exhaustiveness checks	Prevents missing cases in <b>switch</b>
✓ Avoid optional discriminants	They make narrowing harder
✓ Use <b>as const</b> if initializing objects inline	Ensures literal types

---

## Why Not in JavaScript?

Discriminated unions are not runtime constructs — they are **type-level only**, so they don't exist in vanilla JavaScript. However, you can write the same object structures in JS and document them using JSDoc + TypeScript.

---

## Exhaustiveness Check with **never**

```
function handleShape(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return shape.radius;
    case "square":
      return shape.sideLength;
    default:
      // Make sure all types are handled
      const _exhaustive: never = shape;
      return _exhaustive;
  }
}
```

```
}  
}
```

---

## Summary

Feature	Benefit
Discriminated Unions	Type-safe modeling of variants
Common <b>kind/type</b>	Enables type narrowing
Exhaustiveness checks	Ensures future-proof logic
Used in Reducers, APIs	Real-world application ready

---

Would you like a **practice project** that uses discriminated unions, like a mini game or CLI tool simulation?



**Type Assertions as Type, <> Manually tell the compiler what type something is.**

Absolutely! Type assertions in **TypeScript** are a powerful feature that allow you to manually tell the compiler what the type of a value is. This is especially useful when TypeScript can't infer the type correctly but **you, the developer, know more about the value than the compiler does.**

---



## Concept: Type Assertions

There are **two ways** to write type assertions in TypeScript:



### 1. **as Syntax (recommended)**

```
const value = "hello" as string;
```



### 2. **Angle-bracket Syntax <> (not allowed in .tsx files)**

```
const value = <string>"hello";
```

---

## When to Use Type Assertions

1. DOM Manipulation
  2. JSON Parsing
  3. Narrowing union types
  4. Working with `unknown` or `any`
  5. Overriding inferred types
- 

## Beginner Examples

### 1. Simple Type Assertion

```
let someValue: any = "I am a string";
```

```
// Tell TypeScript it's a string to access string methods
let strLength: number = (someValue as string).length;
```

```
console.log(strLength); // Output: 13
```

### 2. Angle-Bracket Syntax

```
let count: any = 42;
let numericCount = <number>count;
```

```
console.log(numericCount + 10); // Output: 52
```

---

## Intermediate Examples

### 3. DOM Element Type Assertion



```
const inputElement = document.getElementById("username") as HTMLInputElement;

inputElement.value = "Routh";
```

#### 4. JSON Parse with Type Assertion

```
interface User {
  name: string;
  age: number;
}

const json = '{"name": "Kiran", "age": 25}';
const user = JSON.parse(json) as User;

console.log(user.name); // Kiran
```

#### 5. Assertion from **unknown** type

```
function getValue(): unknown {
  return "some text";
}

let val = getValue() as string;
console.log(val.length); // 9
```

---

### Advanced Examples

#### 6. Narrowing Union Types

```
function printLength(input: string | number) {
  // Tell TypeScript to treat input as string
  const len = (input as string).length;
  console.log(len); // Runtime error if input is actually a number
}
```

🔥 Use with caution! Better to use type guards instead.

#### 7. Type Assertion with Interfaces

```
interface Admin {
  role: string;
```

```
}  
  
interface User {  
  name: string;  
}  
  
const user = { name: "Kiran", role: "admin" } as User as Admin;  
  
console.log(user.role); // "admin"
```

---

## Important Notes

- Type assertions **do not change** the runtime behavior.
  - They **do not perform type checking or restructuring**.
  - Use only when **you are sure** of the type.
- 

## Playground Tip

Try this in a TypeScript playground:

```
let unknownValue: any = "Hello Typescript";  
  
// Compiler has no idea what type it is  
console.log((unknownValue as string).toUpperCase()); // "HELLO TYPESCRIPT"
```

---

Would you like this in a **TypeScript cheatsheet PDF** or **interactive quiz** for learning?

# Compiler Features `tsconfig.json`, strict mode options, etc. Configure how TypeScript checks and compiles code.

Absolutely, Routh! Let's deep dive into **TypeScript Compiler Features**, focusing on `tsconfig.json` and **strict mode options**, with clear beginner-to-advanced examples.

---

## 1. What is `tsconfig.json`?

`tsconfig.json` tells the TypeScript compiler how to **compile your code** — think of it as the blueprint for your TypeScript project.

---

## Basic `tsconfig.json` Example

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "outDir": "./dist"
  },
  "include": ["src"],
  "exclude": ["node_modules"]
}
```

### What this does:

- `"target": "ES6"` – compile to ECMAScript 6.
- `"module": "commonjs"` – use CommonJS module system (Node.js default).
- `"strict": true` – enable all strict type-checking options.
- `"outDir": "./dist"` – compiled JS goes to `dist` folder.


- `"include": ["src"]` – only compile files in the `src` folder.
- 

## **strict Mode Options Explained with Examples**

When `"strict": true` is enabled, it turns on all the following options:

### 1. **strictNullChecks**

```
let name: string = "Routh";  
// name = null; // ❌ Error: Type 'null' is not assignable to type 'string'
```

 Helps avoid null/undefined bugs by forcing you to check them explicitly.

---

### 2. **noImplicitAny**

```
function greet(user) {  
  return "Hello " + user;  
}  
// ❌ Error: Parameter 'user' implicitly has an 'any' type.
```

 Fix:

```
function greet(user: string) {  
  return "Hello " + user;  
}
```

---

### 3. **strictFunctionTypes**

```
type FuncA = (a: number) => void;  
type FuncB = (a: number | string) => void;
```


```
let fnA: FuncA;  
let fnB: FuncB;
```

```
fnA = fnB; // ❌ Error with strictFunctionTypes
```

Ensures function parameter compatibility for safer code.

---

#### 4. **strictPropertyInitialization**

```
class User {  
  name: string; //  Property 'name' has no initializer  
  
  constructor() {  
    // this.name not initialized  
  }  
}
```

 Fix:


```
class User {  
  name: string;  
  
  constructor() {  
    this.name = "Routh";  
  }  
}
```


---

#### 5. **alwaysStrict**

Automatically adds `"use strict";` in all compiled JS files.

#### 6. **noImplicitThis**


```
function show() {  
  console.log(this); //  Error: 'this' implicitly has type 'any'  
}
```

 Fix with proper context:

```
const obj = {  
  name: "Routh",  
  show(this: { name: string }) {  
    console.log(this.name);  
  }  
};
```

---

## 7. useUnknownInCatchVariables

```
try {  
  throw "error";  
} catch (e) {  
  console.log(e.toUpperCase()); //  e is of type 'unknown'  
}
```

 Fix:

```
try {  
  throw "error";  
} catch (e: any) {  
  console.log(e.toUpperCase());  
}
```

---

## Pro Tip: Enable Strict Mode One-by-One

If you're new to strict typing, you can enable options **individually**:

```
"strictNullChecks": true,  
"noImplicitAny": true
```

---

## Advanced **tsconfig.json** Setup

```
{  
  "compilerOptions": {  
    "target": "ES2020",  
    "module": "ESNext",  
    "moduleResolution": "node",  
    "outDir": "build",  
    "sourceMap": true,  
    "strict": true,  
    "esModuleInterop": true,  
    "forceConsistentCasingInFileNames": true,  
    "skipLibCheck": true,  
  }  
}
```

```
"resolveJsonModule": true
},
"include": ["src/**/*"],
"exclude": ["node_modules", "tests"]
}
```

## Additional Features:

- `"sourceMap"`: Useful for debugging in VSCode.
- `"esModuleInterop"`: Allows mixing default and named imports.
- `"resolveJsonModule"`: Lets you import `.json` files.
- `"forceConsistentCasingInFileNames"`: Avoid case-sensitive issues on different OSes.

---

## Summary Table of Common Compiler Flags

Option	Purpose	Recommended
<code>strict</code>	Enables all strict checks	✓ Yes
<code>noImplicitAny</code>	Forces you to type everything	✓ Yes
<code>strictNullChecks</code>	Prevents null/undefined errors	✓ Yes
<code>outDir</code>	Sets output directory	✓ Yes
<code>sourceMap</code>	Enables debugging in browser/VSCode	✓ Yes
<code>esModuleInterop</code>	Better interop with CommonJS/ESM	✓ Yes
<code>resolveJsonModule</code>	Allows <code>.json</code> imports	✓ Yes
<code>skipLibCheck</code>	Speeds up compilation by skipping <code>node_modules</code> types	Optional

`moduleResolution:` Matches Node.js behavior  
`"node"`

☒ Yes

---

## Practice Challenge for You

 Create a `tsconfig.json` with the following:

- `"strict": true`
- Output goes to `build/`
- Accept imports from `.json` files
- Only include files in `src/`

Then create a `src/index.ts`:

```
import data from "../data.json";  
  
console.log("Loaded data:", data);
```

Create a `data.json`:

```
{  
  "username": "routhkiran",  
  "isActive": true  
}
```

Compile and run with:

```
tsc  
node build/index.js
```

---

If you'd like, I can generate a complete example project folder structure with sample code.  
Would you like that?





## Overloads      Function overloads based on input type      Not possible in plain JS.

Absolutely, Routh Kiran Babu! Let's break down **Function Overloads** in **TypeScript** — a powerful feature that allows you to write different function signatures for different input types (something not possible in plain JavaScript).

---



## What is Function Overloading?

**Function Overloading** lets you define **multiple function signatures** with **different input and return types**, but implement **only one actual function**.

---



## Real-world Scenario

Imagine a function that:

- Returns the length of a **string**
  - Returns the length of an **array**
  - Throws error for anything else
- 



## Step-by-step: Basic to Advanced

---



### 1. Basic Overload Example (string or number)

```
function combine(a: string, b: string): string;  
function combine(a: number, b: number): number;
```

```
function combine(a: any, b: any): any {  
  return a + b;  
}
```

```
console.log(combine(10, 20));    // 30
```

```
console.log(combine("Hi ", "TS")); // Hi TS
// console.log(combine(10, "TS")); // ❌ Error: No matching overload
```

🔑 The implementation is **only one**, but the **overload signatures** allow type checking.

---

## 🟡 2. Array and String Overload

```
function getLength(input: string): number;
function getLength(input: any[]): number;
```

```
function getLength(input: any): number {
  return input.length;
}
```

```
console.log(getLength("TypeScript")); // 10
console.log(getLength([1, 2, 3, 4])); // 4
```

✅ Type-safe: Only **string** or **array** types allowed.

---

## 🟢 3. Object Overload with Custom Types

```
type User = { name: string };
type Admin = { name: string; role: string };
```

```
function describe(person: User): string;
function describe(person: Admin): string;
```

```
function describe(person: any): string {
  if ('role' in person) {
    return `${person.name} is an Admin with role ${person.role}`;
  }
  return `${person.name} is a User`;
}
```

```
console.log(describe({ name: "Kiran" }));
console.log(describe({ name: "Kiran", role: "Tester" }));
```

---

## 4. Overload with Optional Parameters

```
function greet(name: string): string;
function greet(): string;

function greet(name?: string): string {
  return name ? `Hello, ${name}` : "Hello, stranger!";
}

console.log(greet("Routh")); // Hello, Routh
console.log(greet());        // Hello, stranger!
```

---

## 5. Advanced: Return Types Based on Inputs

```
function reverse(input: string): string;
function reverse(input: number[]): number[];

function reverse(input: any): any {
  if (typeof input === "string") {
    return input.split("").reverse().join("");
  } else if (Array.isArray(input)) {
    return input.reverse();
  }
}

console.log(reverse("hello")); // "olleh"
console.log(reverse([1, 2, 3, 4])); // [4, 3, 2, 1]
```

---

## Rules for Overloads in TypeScript

1. **Multiple signatures** but **only one implementation**.
  2. The implementation must handle **all overload cases**.
  3. TypeScript will **enforce correct usage** at compile time.
- 

## Not Possible in Plain JavaScript

In JavaScript, you **can't enforce types**, so function overloading is usually done by checking `typeof` or `arguments.length`, but without type safety.

---

## Summary

Concept	TypeScript Supports	JavaScript Supports
Function Overloads	✓ Yes	✗ No (manual logic)
Type Inference	✓ Yes	✗ No
Compile-Time Errors	✓ Yes	✗ No

---

Would you like to practice these examples in a TypeScript playground or VS Code snippet template? I can generate that for you.