TYPESCRIPT

Fundamentals

Why TypeScript & Setup · Basic Types & Type System
Objects, Arrays & Tuples · Interfaces & Type Aliases
Classes & OOP TypeScript · Generics Fundamentals
Advanced Types · Functions & Overloads · Enums & Literal Types

Philip Osztromok

Generated with Claude

Table of Contents

TypeScript Fundamentals — 9 Chapters

🔵 Why TypeScript & Setup

TypeScript transforms JavaScript with static typing and powerful tooling. In this chapter, we'll explore why TypeScript exists, what problems it solves, and get your development environment ready for the journey ahead.

The Problem TypeScript Solves

JavaScript is incredibly flexible — but that flexibility comes with a cost:

JavaScript vs TypeScript

JavaScript

Dynamic types. Errors appear at runtime:

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

// ❌ No error until runtime!
greet(42);  // TypeError: name.toUpperCase is not a function
```

TypeScript

Static types. Errors caught during development:

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

// ✅ Error caught immediately!
greet(42);  // ❌ Argument of type 'number' is not assignable to parameter of type 'string'
```

Key Benefits of TypeScript

- **Catch bugs early:** Type errors are discovered during development, not in production.
- **Better IDE support:** Autocomplete, refactoring, and navigation all improve dramatically.
- **Self-documenting code:** Types tell developers what functions expect and return.
- **Scalability:** Large codebases stay maintainable as complexity grows.
- **Confidence in refactoring:** Change code knowing the compiler will catch breaking changes.
- **Industry standard:** Used by React, Angular, Vue, Node.js frameworks, and more.

⚙️ Installation & Setup

TypeScript runs on Node.js. If you don't have Node.js installed, download it from nodejs.org (LTS version recommended).

Step 1: Install TypeScript Globally (Optional)

```
npm install -g typescript
```

This makes the `tsc` (TypeScript compiler) available anywhere in your terminal.

Step 2: Create a Project Directory

```
mkdir my-typescript-project
cd my-typescript-project
```

Step 3: Initialize Node.js Project

```
npm init -y
```

This creates a `package.json` file to manage your project's dependencies.

Step 4: Install TypeScript Locally

```
npm install --save-dev typescript
```

Installing locally (rather than globally) is best practice for team projects.

Step 5: Generate tsconfig.json

```
npx tsc --init
```

This creates the `tsconfig.json` configuration file. Review it — the defaults are usually fine for beginners.

⚙️ Understanding tsconfig.json

The `tsconfig.json` file tells the TypeScript compiler how to behave. Here are the most important settings:

```
{
  "compilerOptions": {
    "target": "ES2020",        // JavaScript version to compile to
"module": "commonjs",       // Module system (commonjs for Node.js)
"lib": ["ES2020"],          // Built-in types available
"outDir": "./dist",         // Where compiled .js files go
"rootDir": "./src",         // Where your .ts files are
"strict": true,             // Enable strict type checking
"esModuleInterop": true,    // Better CommonJS compatibility
"skipLibCheck": true,       // Skip type checking of declaration files
"forceConsistentCasingInFileNames": true
  },
  "include": ["src"],         // Include src directory
"exclude": ["node_modules"]  // Exclude node_modules
}
```

🚀 Your First TypeScript Program

Step 1: Create src directory and file

```
mkdir src
// Create: src/hello.ts
```

Step 2: Write TypeScript

```
const message: string = "Hello, TypeScript!";
console.log(message);
```

Step 3: Compile to JavaScript

```
npx tsc
```

This compiles all TypeScript files in `src/` to JavaScript in `dist/`.

Step 4: Run the compiled code

```
node dist/hello.js
```

Output: `Hello, TypeScript!`

💻 Coding Challenges

Challenge 1: Type the Parameters

Add type annotations to this function so TypeScript prevents passing wrong types:

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

Goal: Make sure `add("5", 3)` produces a TypeScript error.

→ Solution

Challenge 2: Debug the tsconfig

You've installed TypeScript, but `npx tsc` gives an error saying it can't find `tsconfig.json`. What command did you forget?

Goal: Recall the command to generate `tsconfig.json`.

→ Solution

Challenge 3: Compile and Run

Create a TypeScript file that prints your name in uppercase. Compile it and run the resulting JavaScript file.

Goal: Demonstrate the full workflow: write TS → compile → run JS.

→ Solution

💡 Real-World Tip: Watch for Compilation Errors

The TypeScript compiler is your first line of defense. If it complains about a type error, stop and read the message carefully — it's trying to protect you from bugs! As you learn, you'll develop a sense for why the compiler is rejecting your code, and you'll start writing types that prevent mistakes naturally.

🎯 What's Next

In the next chapter, we'll explore TypeScript's type system: primitives, any, unknown, and how type inference works. You'll learn the vocabulary that makes TypeScript powerful.

🔤 Basic Types & Type System

TypeScript's power comes from its type system. In this chapter, we'll explore the primitive types, learn how type annotations work, and understand how TypeScript infers types when you don't explicitly declare them.

The Primitive Types

TypeScript includes seven primitive types from JavaScript, plus two special types:

string

Text values. Can use single quotes, double quotes, or backticks.

```
const name: string = "Alice";
const greeting: string = 'Hello';
const message: string = `Hi, ${name}`;
```

number

All numeric values: integers, floats, Infinity, NaN.

```
const count: number = 42;
const price: number = 19.99;
const infinity: number = Infinity;
```

boolean

Only two values: true or false.

```
const isActive: boolean = true;
const hasError: boolean = false;
```

null & undefined

Special values representing absence. Usually used with unions.

```
const x: null = null;
const y: undefined = undefined;
```

bigint

Numbers larger than Number.MAX_SAFE_INTEGER (rare).

```
const huge: bigint = 9007199254740992n;
```

symbol

Unique identifiers (advanced; rarely used in beginner code).

```
const id: symbol = Symbol("id");
```

📝 Type Annotations

A **type annotation** explicitly tells TypeScript what type a variable should be. Use a colon followed by the type name.

```
const age: number = 25;
const name: string = "Bob";
const isStudent: boolean = true;

// ❌ TypeScript error! number is not assignable to string
const city: string = 42;
```

Annotations on Function Parameters & Returns

You can annotate parameters and specify what a function returns:

```
function multiply(x: number, y: number): number {
  return x * y;
}

// ✅ Works
multiply(3, 4);  // Returns 12
// ❌ Error! Argument of type 'string' is not assignable to parameter of type 'number'
multiply("3", 4);
```

🧠 Type Inference

You don't always need to write type annotations. TypeScript can often **infer** the type based on the value you assign.

✅ **No annotation needed:**

```
const count = 42;  // TypeScript knows count is a number
const name = "Alice";  // TypeScript knows name is a string
const active = true;  // TypeScript knows active is a boolean
```

However, inference has limits. Sometimes you need explicit annotations:

❌ **Inference isn't smart enough here:**

```
const data = null;  // TypeScript infers type 'null', might not be what you want
function process(value) {  // No annotation, value's type is unclear
return value.toUpperCase();  // Error: TypeScript doesn't know what value is
}
```

Best Practice: When to Annotate

Situation	Recommendation
Clear from assignment (e.g., `const x = 5`)	Skip annotation, let inference work
Function parameters	**Always annotate**
Function return values	Highly recommended (documents intent)
Complex/unclear types	Always annotate for clarity
Variable initialized to `null` or `undefined`	Always annotate (inference too vague)

🎭 Special Types: any & unknown

The any Type

`any` is an "escape hatch" that disables type checking. Use it sparingly!

```
let data: any = "hello";
data = 42; // ✅ Fine
data = true; // ✅ Fine
data.toUpperCase(); // ✅ Fine (no error, even though data might be a number!)
```

Why avoid `any`? It defeats the entire purpose of TypeScript. You lose type safety and IDE support.

The unknown Type

`unknown` is the safer, stricter cousin of `any`. It's type-safe because it forces you to check what the value actually is before using it.

```
let data: unknown = "hello";
data = 42; // ✅ Fine
// ❌ Error! Object is of type 'unknown'
data.toUpperCase();

// ✅ But this is fine — we checked first!
if (typeof data === "string") {
  data.toUpperCase(); // Now TypeScript knows data is a string
}
```

any vs unknown

> **Use `unknown` instead of `any`.** Always. `unknown` requires you to check the type before using the value, which keeps your code safe.

🔍 Type Narrowing (Preview)

As we saw with `unknown`, TypeScript can narrow types based on your code. This happens with:

- **typeof checks:** `typeof x === "string"`
- **Instanceof checks:** `x instanceof Date`
- **Truthiness checks:** `if (x) { ... }`
- **Equality checks:** `if (x === null) { ... }`

We'll dive deep into type narrowing in a later chapter, but understanding it now will help you write safer TypeScript.

💻 Coding Challenges

Challenge 1: Type Annotation Errors

Fix the type errors in this code by adding correct type annotations:

```
const age = "25";
const score = 98.5;
function greet(person) {
  return "Hello, " + person;
}
```

Goal: Add type annotations so each variable/parameter has the correct type.

→ Solution

Challenge 2: Inference vs Annotation

Which of these variables need explicit type annotations, and which can rely on inference?

```
const x = 42;
const y = null;
function process(value) { ... }
const isReady = true;
```

Goal: Identify which ones need annotations and explain why.

→ Solution

Challenge 3: any vs unknown

Write a function that safely handles `unknown` input. Check if it's a number, and if so, double it. Otherwise, return 0.

```
function doubleIfNumber(value: unknown): number {
  // Your code here
}
```

Goal: Demonstrate type narrowing with `unknown`.

→ Solution

💡 Real-World Tip: Be Explicit with Functions

While you can let TypeScript infer types for variables, always be explicit with function parameters and return types. This makes your code self-documenting and easier for others (and future you!) to understand. A 10-second type annotation saves 10 minutes of confusion later.

🎯 What's Next

Now that you understand primitives and type annotations, we'll explore how to work with collections: objects, arrays, and tuples. You'll learn how to describe the shape of complex data structures.

📦 Objects, Arrays & Tuples

Primitives are the building blocks, but real programs work with collections: objects holding multiple properties, arrays of values, and tuples with fixed structures. This chapter teaches you to type these complex data structures with precision.

🏠 Object Types

Basic Object Type Annotation

Objects in TypeScript are typed by listing their properties and types:

```
const person: { name: string; age: number } = {
  name: "Alice",
  age: 30
};

// ✅ Works
console.log(person.name);  // "Alice"
// ❌ Error! Property 'email' does not exist
console.log(person.email);
```

Optional Properties

Use a question mark (`?`) to mark a property as optional:

```
const user: { name: string; email?: string } = {
  name: "Bob"
// email is optional, so it's fine to omit it
};

// ✅ Both work:
const user2 = { name: "Charlie", email: "charlie@example.com" };
const user3 = { name: "Diana" };  // No email property
```

Readonly Properties

Use `readonly` to prevent a property from being changed after creation:

```
const config: { readonly apiKey: string; port: number } = {
  apiKey: "secret123",
  port: 3000
};

config.port = 4000;  // ✅ Fine, port is mutable
config.apiKey = "newsecret";  // ❌ Error! Cannot assign to readonly property
```

Nested Objects

Objects can contain other objects:

```
const company: {
  name: string;
  address: {
    city: string;
    zipCode: number;
  }
} = {
  name: "TechCorp",
  address: {
    city: "San Francisco",
    zipCode: 94105
  }
};

console.log(company.address.city);  // "San Francisco"
```

📋 Arrays

Array Type Syntax

Arrays are typed by specifying the type of their elements. Use either syntax:

Type[]

Most common syntax:

```
const numbers: number[] = [1, 2, 3];
const names: string[] = ["Alice", "Bob"];
const flags: boolean[] = [true, false];
```

Array<Type>

Generic syntax (less common):

```
const numbers: Array<number> = [1, 2, 3];
const names: Array<string> = ["Alice", "Bob"];
```

Type Safety with Arrays

TypeScript catches type mismatches in arrays:

```
const scores: number[] = [95, 87, 92];

// ✅ Works - adding a number
scores.push(88);

// ❌ Error! Argument of type 'string' is not assignable to parameter of type 'number'
scores.push("100");

// ✅ IDE knows scores[0] is a number
const firstScore = scores[0];  // Type: number
```

Arrays of Objects

Combine array and object types for realistic data structures:

```
const users: { name: string; age: number }[] = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 35 }
];

// ✅ TypeScript knows users[0] is an object with name and age
console.log(users[0].name);  // "Alice"
```

🎯 Tuples

A **tuple** is an array with a **fixed length** and **specific types at each position**. Unlike regular arrays, tuples enforce the exact structure.

Basic Tuple Syntax

```
const point: [number, number] = [10, 20];
const rgb: [number, number, number] = [255, 128, 0];
const response: [string, number] = ["success", 200];

// ❌ Error! Type '[number, number, string]' is not assignable to type '[number, number]'
const badPoint: [number, number] = [10, 20, "invalid"];

// ❌ Error! Length mismatch
const anotherBad: [number, number] = [10];
```

Tuples with Optional Elements

Use `?` to make tuple elements optional:

```
const maybe: [string, number?] = ["hello"];  // ✅ Fine
const also: [string, number?] = ["hello", 42];  // ✅ Also fine
```

Tuples with Named Elements

For clarity, you can name tuple elements:

```
const namedTuple: [x: number, y: number] = [5, 10];
const response: [status: string, code: number] = ["OK", 200];

// Names make the code self-documenting:
console.log(`Response: ${response.status} (${response.code})`);
```

Readonly Tuples

Prevent tuple elements from being modified:

```
const coordinates: readonly [number, number] = [5, 10];

// ❌ Error! Cannot assign to readonly property
coordinates[0] = 20;
```

🔀 Union Types (Preview)

Sometimes a value can be one of several types. Use the pipe operator (`|`) to specify multiple options. We'll dive deep into unions later, but you'll encounter them often.

```
const id: string | number = 42;  // ✅ Can be string or number
id = "user-123";  // ✅ Still valid
const values: (string | number)[] = [1, "two", 3];  // Array of mixed types
```

💻 Coding Challenges

Challenge 1: Type a Product Object

Create a type annotation for a product object with the following properties:

- name (string)
- price (number)
- inStock (boolean)
- description (string, optional)

Goal: Write the type annotation and create an object that matches it. Test with and without the optional property.

→ Solution

Challenge 2: Type Safety with Arrays

Given an array of student scores, write a function that:

- Takes an array of numbers (scores)
- Returns a number (the average)
- Type annotations required for parameters and return value

Goal: Demonstrate array type safety and function annotations together.

→ Solution

Challenge 3: Tuple for Coordinates

Create a tuple type for a 2D point with (x, y) coordinates. Then create three constants:

- origin at (0, 0)
- pointA at (5, 10)
- pointB at (15, 20)

Goal: Demonstrate tuple creation and fixed-length constraints.

→ Solution

💡 Real-World Tip: Know When to Use Tuples

Tuples are great for fixed structures like API responses ([status, data]), coordinates, or return values from destructuring. But for larger objects with multiple properties, use object types (or interfaces — coming in Chapter 4). Tuples scale poorly with complexity, while objects stay readable as you add properties.

🎯 What's Next

Now that you can type objects and arrays, we'll learn **Interfaces** — the TypeScript way to define reusable object shapes and create contracts that multiple objects can implement.

🤝 Interfaces & Type Aliases

So far, you've written type annotations inline. As your code grows, repeating the same object shape becomes tedious and error-prone. This chapter introduces two ways to name and reuse types: **type aliases** and **interfaces**. Both solve the problem—knowing when to use each is the art.

🏷️ Type Aliases

A **type alias** gives a name to any type using the `type` keyword. It's like assigning a type to a variable.

Basic Type Alias

```
type Age = number;

const myAge: Age = 30; // ✅ Works
const yourAge: Age = "thirty"; // ❌ Error! Type 'string' is not assignable to type 'number'
```

Type Alias for Objects

The real power: reuse object shapes across your codebase:

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

const user1: User = { name: "Alice", email: "alice@example.com", age: 30 };
const user2: User = { name: "Bob", email: "bob@example.com", age: 25, isActive: true };

// ✅ Both match the User type
greet(user1);
greet(user2);
```

Union Type Aliases

Type aliases shine with unions — combining multiple types:

```
type ID = string | number;
type Status = "pending" | "active" | "inactive";

const userId: ID = 42; // ✅ number is OK
const username: ID = "user123"; // ✅ string is OK
const currentStatus: Status = "active"; // ✅ Valid status
const badStatus: Status = "deleted"; // ❌ Not a valid status option
```

📋 Interfaces

An **interface** is specifically for describing the shape of objects. It's similar to a type alias for objects, but with some key differences (which we'll explore).

Basic Interface

```
interface User {
  name: string;
  email: string;
  age: number;
  isActive?: boolean;
}

const user: User = { name: "Alice", email: "alice@example.com", age: 30 };
```

Notice the syntax difference: no `=` sign, no curly braces required.

Extending Interfaces

Interfaces can inherit from other interfaces using `extends`:

```
interface User {
  name: string;
  email: string;
}

interface AdminUser extends User {
  role: "admin";
  permissions: string[];
}

// AdminUser now has name, email, role, and permissions
const admin: AdminUser = {
  name: "Alice",
  email: "alice@example.com",
  role: "admin",
  permissions: ["read", "write", "delete"]
};
```

Merging Interfaces

A unique power of interfaces: you can declare the same interface multiple times and they merge:

```
interface User {
  name: string;
}

interface User {
  age: number;
}

// User now has both name AND age
const user: User = { name: "Alice", age: 30 };
```

This is powerful for adding properties to types across different files (think plugins).

⚖️ Type Aliases vs Interfaces

When to Use Each

Use Interfaces For:

- Object shapes (contracts)
- Classes to implement
- Code that merges types
- Public API definitions

Use Type Aliases For:

- Unions (string | number)
- Tuples ([string, number])
- Primitives (numbers, strings)
- Complex compositions

Key Differences

✅ Interfaces can extend

```
interface Admin extends User { ... }
```

✅ Interfaces merge

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

✅ Types support unions

```
type Status = "on" | "off";
```

✅ Types are more flexible

```
type Callback = () => void;
```

🔧 Methods in Interfaces

Interfaces can describe methods (functions) as properties:

```
interface Logger {
  log(message: string): void;
  error(message: string): void;
}

const consoleLogger: Logger = {
  log: (msg) => console.log(msg),
  error: (msg) => console.error(msg)
};
```

🔒 Readonly and Intersection Types

Readonly Properties

```
interface Config {
  readonly apiKey: string;
  port: number;
}

const config: Config = { apiKey: "secret", port: 3000 };
config.port = 4000;  // ✅ Fine
config.apiKey = "newsecret";  // ❌ Error! Cannot assign to readonly
```

Intersection Types (Preview)

Combine multiple types into one using `&`:

```
type Employee = User & { employeeId: number };

// Employee has all properties from User PLUS employeeId
const emp: Employee = {
  name: "Alice",
  email: "alice@example.com",
  age: 30,
  employeeId: 12345
};
```

💻 Coding Challenges

Challenge 1: Create a User Type Alias

Define a type alias for a user with:

- id (number)
- username (string)
- email (string)
- isVerified (boolean, optional)

Goal: Create the type alias and use it to define 2-3 user objects.

→ Solution

Challenge 2: Extend an Interface

Create a base `Person` interface with name and age, then create an `Employee` interface that extends it and adds:

- employeeId (number)
- department (string)

Goal: Demonstrate interface inheritance.

→ Solution

Challenge 3: Union Type for Status

Create a `Status` type that can only be one of: "pending", "approved", or "rejected". Then create a function that accepts this type and logs the status.

```
function handleStatus(status: Status): void {
  // Your implementation
}
```

Goal: Demonstrate union types with literals.

→ Solution

💡 Real-World Tip: Start with Interfaces

As a beginner, favor interfaces for object shapes and type aliases for everything else. Interfaces are slightly more limited, which forces you to write clearer, more maintainable code. You can always switch to a type alias if you need unions or other advanced features.

🎯 What's Next

Now that you can define contracts with types and interfaces, Chapter 5 introduces **Classes & Object-Oriented TypeScript**, where you'll see interfaces in action as contracts that classes implement.

🏗️ Classes & Object-Oriented TypeScript

Classes are the core of object-oriented programming. They let you create blueprints for objects, define behavior with methods, enforce constraints with access modifiers, and share functionality through inheritance. TypeScript adds type safety to JavaScript's class system.

🏛️ Basic Class Structure

Your First Class

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

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

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

// Create an instance
const myDog = new Dog("Buddy", 3);
myDog.bark();  // Output: Buddy says woof!
```

Key parts:

- `class Dog { }` — defines the blueprint
- `name: string; age: number;` — properties (with types!)
- `constructor()` — runs when you create a new instance
- `bark()` — a method (function inside the class)
- `this` — refers to the current instance
- `new Dog()` — creates an instance

🔐 Access Modifiers

Access modifiers control who can access properties and methods. TypeScript provides three: `public`, `private`, and `protected`.

public

Default. Anyone can access from anywhere.

```
class User {
  public name: string;
}
const user = new User();
user.name = "Alice";  // ✅ OK
```

private

Only accessible inside the class.

```
class User {
  private password: string;
}
const user = new User();
console.log(user.password);  // ❌ Error!
```

protected

Accessible inside the class and subclasses.

```
class Animal {
  protected health: number;
}
class Dog extends Animal {
  heal() { this.health++; }  // ✅ OK
}
```

readonly

Can't be changed after initialization.

```
class User {
 readonly id: number;
 constructor(id: number) {
  this.id = id;  // ✅ OK in constructor
 }
}
const user = new User(1);
user.id = 2;  // ❌ Error! readonly
```

Why Access Modifiers Matter

Consider a bank account:

```
class BankAccount {
 private balance: number = 0;

 deposit(amount: number): void {
  if (amount > 0) this.balance += amount;
 }

 withdraw(amount: number): boolean {
  if (amount <= this.balance) {
   this.balance -= amount;
   return true;
  }
  return false;
 }

 public getBalance(): number {
  return this.balance;
 }
}

const account = new BankAccount();
account.deposit(100);  // ✅ OK - uses method
console.log(account.getBalance());  // ✅ OK - public method
account.balance = 1000000;  // ❌ Error! balance is private
```

✅ **Private balance enforces business logic:**

You can't directly set balance to an invalid amount. You must use deposit/withdraw, which validate the transaction.

👫 Inheritance

Inheritance lets you create a class that extends another, inheriting its properties and methods while adding its own.

```
class Animal {
  name: string;

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

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

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

  // Override the parent method
move(): void {
    console.log(`${this.name} runs fast`);
  }
}

const dog = new Dog("Buddy");
dog.move();  // Output: Buddy runs fast (overridden method)
dog.bark();  // Output: Buddy says woof!
```

super: Calling Parent Methods

Use `super` to call parent class methods:

```
class Dog extends Animal {
  breed: string;

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

  move(): void {
    super.move();  // Call parent method
    console.log(` (as a ${this.breed})`);
  }
}
```

🎭 Abstract Classes

An **abstract class** is a blueprint that can't be instantiated directly. Its purpose is to define methods that subclasses must implement. Think of it as a contract.

```
abstract class Shape {
  abstract getArea(): number;  // Subclasses MUST implement
public describe(): void {
    console.log(`This shape has area: ${this.getArea()}`);
  }
}

// ❌ Can't do this:
const shape = new Shape();  // Error! Can't instantiate abstract class
class Circle extends Shape {
  radius: number;

  constructor(radius: number) {
    super();
    this.radius = radius;
  }

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

// ✅ This works:
const circle = new Circle(5);
circle.describe();  // Output: This shape has area: 78.539...
```

⚡ Constructor Parameter Shorthand

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

```
// Before (verbose):
class User {
  name: string;
  age: number;
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
}

// After (shorthand):
class User {
  constructor(
    public name: string,
    public age: number
  ) {}
}

// Both are equivalent!
```

💻 Coding Challenges

Challenge 1: Create a Vehicle Class

Create a `Vehicle` class with:

- Properties: brand, model, year (all public)
- Constructor to initialize these
- A method `getInfo()` that returns a description string

Goal: Create 2-3 vehicle instances and call getInfo().

→ Solution

Challenge 2: Access Modifiers

Create a `BankAccount` class with:

- Private `balance` property (starts at 0)
- Public `deposit(amount)` method
- Public `getBalance()` method
- Ensure balance can't go negative

Goal: Demonstrate why private properties protect data.

→ Solution

Challenge 3: Inheritance and Overriding

Create an `Animal` class with a `speak()` method. Then create `Dog` and `Cat` subclasses that override `speak()` with their own sounds.

- Animal: speaks generically
- Dog: says "Woof!"
- Cat: says "Meow!"

Goal: Demonstrate polymorphism and method overriding.

→ Solution

💡 Real-World Tip: Use private by Default

Start by making properties `private`, then expose what you need through public methods. This is safer than making everything public and trying to restrict later. It forces you to think about your API and protects internal state from accidental changes.

🎯 What's Next

Classes are powerful, but sometimes you need more flexibility. Chapter 6 introduces **Generics** — the TypeScript way to write code that works with many types while staying type-safe.

🔤 Generics Fundamentals

Generics are TypeScript's way of writing flexible, reusable code that works with many types while maintaining type safety. Imagine a function that works with numbers, strings, objects—any type—and still catches type errors. That's generics.

⚠️ The Problem Generics Solve

Without generics, you'd write the same function multiple times for different types, or resort to `any` and lose type safety.

```
// ❌ Without generics: lots of duplication
function getFirstString(arr: string[]): string {
  return arr[0];
}

function getFirstNumber(arr: number[]): number {
  return arr[0];
}

function getFirstAny(arr: any[]): any {
  return arr[0];
}

// Same logic, written 3 times. And if you need boolean, object, etc., write it again!
```

❌ With `any`, you lose type safety:

```
const str = getFirstAny(["hello", "world"]);
// TypeScript thinks str is 'any', so:
str.toUpperCase();  // ✅ Works, but TypeScript doesn't know if it's safe
str.toReverse();  // ❌ No error, but this method doesn't exist!
```

📝 Generic Functions

Your First Generic Function

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

// TypeScript infers the type based on usage:
const firstString = getFirst(["hello", "world"]);  // T is 'string'
// firstString is type 'string', so:
firstString.toUpperCase();  // ✅ TypeScript knows it's safe
const firstNumber = getFirst([1, 2, 3]);  // T is 'number'
// firstNumber is type 'number', so:
firstNumber.toFixed(2);  // ✅ TypeScript knows it's safe
```

> ✅ **Key insight:**
>
> `<T>` is a **type parameter** — a placeholder for any type. When you call `getFirst([1,2,3])`, TypeScript sees the argument is `number[]`, so it sets `T = number`. The function now behaves as if it were typed specifically for numbers.

Multiple Type Parameters

Functions can have multiple type parameters:

```
function pair<T, U>(first: T, second: U): [T, U] {
  return [first, second];
}

// Different types for each parameter:
const result = pair("hello", 42);  // T is 'string', U is 'number'
// result is ["hello", 42] with type [string, number]
```

🏛️ Generic Classes

Classes can be generic too. A common example is a container or wrapper:

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

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

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

  setValue(newValue: T): void {
    this.value = newValue;
  }
}

// Create a box for strings:
const stringBox = new Box<string>("hello");
stringBox.getValue();  // Type: string
// Create a box for numbers:
const numberBox = new Box<number>(42);
numberBox.getValue();  // Type: number
```

🔒 Generic Constraints

Sometimes you want to restrict what types a generic can accept. For example, "only types that have a `length` property" or "only types that extend a specific interface."

Constraint: Must Have a Property

```
interface HasLength {
  length: number;
}

function getLength<T extends HasLength>(item: T): number {
  return item.length;
}

// ✅ Works: strings have length
getLength("hello");  // Returns 5
// ✅ Works: arrays have length
getLength([1, 2, 3]);  // Returns 3
// ❌ Error: numbers don't have length
getLength(42);  // Error!
```

Constraint: Must Extend a Type

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

// ✅ Works: objects can be merged
merge({ a: 1 }, { b: 2 });

// ❌ Error: strings can't be merged this way
merge("hello", "world");
```

Constraint: Generic Must Be a Key

Ensure a type parameter is a key of another type parameter:

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

const person = { name: "Alice", age: 30 };

getProperty(person, "name");  // ✅ "name" is a key of person
getProperty(person, "email");  // ❌ "email" is not a key of person
```

🎯 Common Generic Patterns

Array Methods

```
function map<T, U>(arr: T[], fn: (T) => U): U[] {
  return arr.map(fn);
}

// Transform strings to numbers:
const lengths = map(["a", "bb", "ccc"], (s) => s.length);
// Result: [1, 2, 3]
```

Promise Resolution

```
function fetchData<T>(url: string): Promise<T> {
  return fetch(url).then((res) => res.json());
}

interface User {
  id: number;
  name: string;
}

// Promise resolves to User:
const userPromise = fetchData<User>("/api/user");
// userPromise is Promise<User>
```

💻 Coding Challenges

Challenge 1: Generic Array Reverse

Write a generic function that reverses an array of any type and returns the reversed array. The function should maintain type safety.

```
function reverse<T>(arr: T[]): T[] {
  // Your implementation
}
```

Goal: Work with strings, numbers, objects—any type.

→ Solution

Challenge 2: Generic Cache Class

Create a generic `Cache<T>` class that stores a value and provides get/set methods. The value can be any type.

- Constructor takes an initial value
- `get()` method returns the cached value
- `set(value)` method updates the cache

Goal: Demonstrate generic classes.

→ Solution

Challenge 3: Generic with Constraint

Write a generic function that accepts an object and a property name, then returns the property value. Use a constraint to ensure the property exists on the object.

```
function getProp<T, K>(obj: T, key: K) {
  // Your implementation
}
```

Goal: Demonstrate generic constraints with `keyof`.

→ Solution

💡 Real-World Tip: Generics Make Your Code DRY

If you find yourself writing the same function for different types, or using `any` to make it work with multiple types, generics are your answer. Libraries like React, Vue, Express, and database ORMs use generics extensively. Understanding them now will pay dividends as you read real-world code.

🎯 What's Next

Generics are powerful, but TypeScript has even more advanced type features. Chapter 7 covers **Advanced Types** — union types, type guards, discriminated unions, and more sophisticated patterns for keeping your code type-safe.

🔀 Advanced Types

You've seen union types in passing. Now we'll master them. Union types are where TypeScript shines — they let you express that a value can be one of several types, and TypeScript will help you handle each case correctly. This chapter teaches type guards, discriminated unions, and exhaustiveness checking.

🔀 Union Types Deep Dive

Basic Unions

A union type says "this can be one of these types":

```
type ID = string | number;

function printID(id: ID) {
  console.log(id);
}

printID(42);  // ✅ Works: number is OK
printID("user-123");  // ✅ Works: string is OK
printID(true);  // ❌ Error! boolean is not in the union
```

The Problem: Accessing Properties

With unions, TypeScript only knows about properties that exist on ALL union members:

```
type Padded = string | number;

function stringify(x: Padded) {
  // ❌ Error! Property 'toUpperCase' does not exist on type 'number'
return x.toUpperCase();
}

// Problem: toUpperCase exists on strings but NOT on numbers
// So TypeScript forbids it to prevent runtime errors
```

> ✅ **Solution: Type Guards**
>
> Check what type the value actually is, then use it accordingly.

🛡️ Type Guards

A **type guard** is code that checks the type of a variable and narrows it to a more specific type. TypeScript then knows you've verified the type and allows you to use type-specific properties.

typeof Guard

```
type Padded = string | number;

function stringify(x: Padded) {
  // typeof guard: check if it's a string
if (typeof x === "string") {
    return x.toUpperCase();  // ✅ Now TypeScript knows it's a string
  }
  // Here, x must be number
return x.toFixed(2);  // ✅ Now TypeScript knows it's a number
}
```

instanceof Guard

For classes, use `instanceof`:

```
class Cat {
  meow() { console.log("Meow!"); }
}

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

function petSound(pet: Cat | Dog) {
  if (pet instanceof Cat) {
    pet.meow();  // ✅ TypeScript knows it's a Cat
  } else {
    pet.bark();  // ✅ TypeScript knows it's a Dog
  }
}
```

Truthiness Guard

JavaScript truthiness checks narrow types:

```
function printLength(str: string | null) {
  if (str) {
    // Inside this block, str cannot be null
    console.log(str.length);  // ✅ Safe
  }
}
```

🎯 Discriminated Unions

A **discriminated union** (also called tagged unions) is a union of types that share a common property—the "discriminator"—which you can use to determine the exact type. This pattern is incredibly powerful and type-safe.

The Pattern

```
type SuccessResponse = {
  status: "success";  // Discriminator: always "success"
  data: string;
};

type ErrorResponse = {
  status: "error";  // Discriminator: always "error"
  error: string;
};

type Response = SuccessResponse | ErrorResponse;

function handleResponse(response: Response) {
  // Check the discriminator (status property)
if (response.status === "success") {
    // TypeScript narrows to SuccessResponse
    console.log("Data:", response.data);  // ✅ 'data' only exists here
  } else {
    // TypeScript narrows to ErrorResponse
    console.log("Error:", response.error);  // ✅ 'error' only exists here
  }
}
```

With switch Statement

Discriminated unions shine with switch statements:

```
type Result =
  | { status: "pending" }
  | { status: "success"; value: number }
  | { status: "error"; message: string };

function handle(result: Result) {
  switch (result.status) {
    case "pending":
      console.log("Waiting...");
      break;
    case "success":
      console.log("Result:", result.value);  // ✅ 'value' available
break;
    case "error":
      console.log("Error:", result.message);  // ✅ 'message' available
break;
  }
}
```

✅ Exhaustiveness Checking

TypeScript can verify you've handled all possible cases in a union. This catches bugs when you add a new type to the union but forget to handle it somewhere.

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; side: number }
  | { kind: "triangle"; side: number };

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.side ** 2;
    // ❌ Error! Not all cases covered (missing 'triangle')
  }
}

// TypeScript error: Function lacks ending return statement and
// return type does not include 'undefined'
```

✅ Fix: Handle all cases

```
case "triangle":
  return (shape.side ** 2) * Math.sqrt(3) / 4;
```

🚫 The never Type

The `never` type represents impossible values. Use it for exhaustiveness checking:

```
function assertNever(x: never): never {
  throw new Error("Should not reach here");
}

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.side ** 2;
    case "triangle":
      return (shape.side ** 2) * Math.sqrt(3) / 4;
    default:
      // If you add a new Shape type and forget to handle it here,
// TypeScript will complain that the new type doesn't match 'never'
return assertNever(shape);
  }
}
```

💻 Coding Challenges

Challenge 1: Type Guard Function

Create a type guard function that checks if a value is a string. Use it in a function that only proceeds if the guard passes.

```
function isString(value: unknown): boolean {
  // Your implementation
}
```

Goal: Demonstrate typeof guard pattern.

→ Solution

Challenge 2: Discriminated Union

Create a discriminated union for API responses with three states: loading, success (with data), and error (with message). Write a function to handle each.

- Define the union type
- Write a handler function using switch statement
- Test all three cases

Goal: Demonstrate discriminated unions and type narrowing.

→ Solution

Challenge 3: Exhaustiveness with never

Create a union of three animal types with a `makeSound()` function. Use the `never` type to ensure all cases are handled.

- Define three animal types (Cat, Dog, Bird)
- Each has different sound methods
- Use assertNever in the default case

Goal: Demonstrate exhaustiveness checking with never type.

→ Solution

💡 Real-World Tip: Discriminated Unions > Many Conditions

Instead of checking multiple boolean flags or scattered properties, use discriminated unions to represent distinct states explicitly. This catches bugs at compile-time and makes code self-documenting. The pattern is used extensively in React (status states), error handling (Result types), and state machines.

🎯 What's Next

We've mastered unions and type narrowing. Chapter 8 covers **Functions & Overloads** — how to type functions that behave differently based on their arguments, and when to use function overloads.

📞 Functions & Overloads

You've written typed functions. Now we'll explore how TypeScript handles functions that can behave differently based on their arguments. Function overloads let you define multiple signatures for a single function, making it type-safe and self-documenting.

🔧 Function Typing Basics (Review)

Parameter Types and Return Types

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

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

// Arrow function syntax
const multiply = (a: number, b: number): number => a * b;

// Optional parameters
function greetOptional(name: string, title?: string): string {
  return title ? `${title} ${name}` : name;
}

// Rest parameters
function sum(...numbers: number[]): number {
  return numbers.reduce((a, b) => a + b, 0);
}
```

🎯 Function Overloads

A function overload lets you define multiple signatures for the same function. The actual implementation comes after all the signatures. TypeScript uses these signatures to type-check calls to the function.

The Pattern

```
// Overload signatures (just the type, no body)
function describe(value: string): string;
function describe(value: number): string;

// Implementation (with union type to match all overloads)
function describe(value: string | number): string {
 if (typeof value === "string") {
  return `String: ${value}`;
 } else {
  return `Number: ${value.toFixed(2)}`;
 }
}

// Callers see the specific signatures
describe("hello"); // ✅ Matches first signature
describe(42);      // ✅ Matches second signature
describe(true);  // ❌ Error! No overload for boolean
```

✅ **Key Point:** The overload signatures are for type-checking. The implementation uses a union type that covers all cases.

Overloads with Different Return Types

```
// Each overload can have a different return type
function process(input: string): string;
function process(input: number): number;
function process(input: boolean): boolean;

function process(input: string | number | boolean) {
  if (typeof input === "string") {
    return input.toUpperCase();
  } else if (typeof input === "number") {
    return input * 2;
  } else {
    return !input;
  }
}

const str: string = process("hello");  // TypeScript knows return is string
const num: number = process(5);      // TypeScript knows return is number
const bool: boolean = process(true);  // TypeScript knows return is boolean
```

Overloads with Optional Parameters

```
// Different number of parameters
function createElement(tag: string): HTMLElement;
function createElement(tag: string, content: string): HTMLElement;
function createElement(tag: string, content: string, className: string): HTMLElement;

function createElement(
  tag: string,
  content?: string,
  className?: string
): HTMLElement {
  const el = document.createElement(tag);
  if (content) el.textContent = content;
  if (className) el.className = className;
  return el;
}

createElement("div");  // ✅ Just tag
createElement("div", "Hello");  // ✅ Tag + content
createElement("div", "Hello", "container");  // ✅ All three
```

🤔 Overloads vs Union Types

When should you use overloads instead of union types? Let's compare:

❌ Union Type (Bad)

```
function padString(
  value: string | number
): string {
  return String(value).padStart(10);
}

const result = padString(42);
// TypeScript doesn't know result is a string
// The caller loses type information!
```

✅ Overloads (Good)

```
function padString(value: string): string;
function padString(value: number): string;

function padString(value: string | number) {
  return String(value).padStart(10);
}

const result = padString(42);
// TypeScript knows result is a string!
// Overloads preserve type information
```

When to Use Overloads

- **Different return types** — Based on input type, return type differs
- **Different parameters** — Accept different number of parameters
- **Complex relationships** — Type of parameter 2 depends on parameter 1
- **API clarity** — Make intent clear to callers

When NOT to Use Overloads

- **Same behavior** — If logic is identical, just use union types
- **Simple cases** — Optional parameters are often enough
- **Too many overloads** — More than 3-4 usually signals design problem

🚀 Advanced Overload Patterns

Overloads with Generics

```
// Generic function with overloads
function getProperty<T>(obj: T, key: string): unknown;
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K];

function getProperty<T>(obj: T, key: string) {
  return obj[key as keyof T];
}

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

const user: User = { name: "Alice", age: 30 };

// ✅ TypeScript knows this returns a string
const name: string = getProperty(user, "name");

// ✅ TypeScript knows this returns a number
const age: number = getProperty(user, "age");
```

Overloads with Conditional Return Types

```
// Using conditional types with overloads
function flatten<T>(arr: T[]): T[];
function flatten<T>(arr: T[][]): T[];

function flatten<T>(arr: T[] | T[][]): T[] {
  return arr.flat() as T[];
}

const single: string[] = flatten(["a", "b"]);
const double: number[] = flatten([[1, 2], [3, 4]]);
```

📋 Best Practices

✅ DO: Overloads for Clearer Intent

When different input types produce different return types, overloads document this relationship:

```
function parse(input: string): Record<string, unknown>;
function parse(input: Buffer): Uint8Array;

function parse(input: string | Buffer) {
  // implementation
}
```

❌ DON'T: Too Many Overloads

More than 3-4 overloads often means your function is doing too much:

```
// ❌ Too many overloads - refactor instead
function process(input: string): string;
function process(input: number): number;
function process(input: boolean): boolean;
function process(input: Date): string;
function process(input: object): object;
```

💡 Tip: Overloads Should Be Simple

If the overload signatures are hard to understand, the function signature might be too complex. Consider breaking it into multiple, simpler functions instead.

💻 Coding Challenges

Challenge 1: Basic Function Overload

Create a function that accepts either a string or a number and returns a boolean. If string, check if length > 5. If number, check if > 100.

- Define two overload signatures
- Write the implementation with both cases
- Test with both string and number inputs

Goal: Practice basic overload syntax and type checking.

→ Solution

Challenge 2: Overload with Different Return Types

Create a `convertToJSON` function that takes an object and optional formatting. If formatting is true, return pretty JSON. If false, return compact JSON.

- Define overloads for formatted/unformatted
- Both return strings (different content)
- Use JSON.stringify with appropriate parameters

Goal: Practice conditional behavior with overloads.

→ Solution

Challenge 3: Generic Overload

Create a `wrapInArray` function that takes a single value OR an array, and always returns an array. If passed a single value, wrap it. If passed an array, return it as-is.

- Define overloads for single value and array
- Use generics to preserve the element type
- Ensure correct array nesting

Goal: Practice generic overloads and type preservation.

→ Solution

🌎 Real-World Example: Event Listener

```
// Realistic addEventListener with proper typing
function addEventListener(
  element: HTMLElement,
  event: "click",
  handler: (e: MouseEvent) => void
): void;

function addEventListener(
  element: HTMLElement,
  event: "input",
  handler: (e: InputEvent) => void
): void;

function addEventListener(
  element: HTMLElement,
  event: "click" | "input",
  handler: (e: Event) => void
) {
  element.addEventListener(event, handler);
}

// Callers get proper type checking for each event
const button = document.querySelector("button")!;
const input = document.querySelector("input")!;

addEventListener(button, "click", (e: MouseEvent) => {
  // ✅ e is typed as MouseEvent
  console.log(e.clientX);
});

addEventListener(input, "input", (e: InputEvent) => {
  // ✅ e is typed as InputEvent
  console.log(e.data);
});
```

🎯 What's Next

We've mastered function overloads and learned when to use them. Chapter 9 covers **Enums & Literal Types** — how to create strongly-typed constants and restrict values to specific options.

📦 Enums & Literal Types

Enums and literal types let you restrict values to a specific set of allowed options. This chapter covers string enums, numeric enums, const enums, and how literal types offer a modern alternative. You'll learn when to use each approach.

📝 String Enums

Basic String Enum

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT"
}

function move(direction: Direction) {
  console.log(`Moving ${direction}`);
}

move(Direction.Up);     // ✅ "Moving UP"
move("UP");         // ❌ Error! String literal not allowed
```

Enum Without Explicit Values

If you don't specify values, TypeScript uses the member name:

```
enum Status {
  Active,    // value is "Active"
  Inactive,  // value is "Inactive"
  Pending    // value is "Pending"
}

const status: Status = Status.Active;
console.log(status);  // "Active"
```

🔢 Numeric Enums

```
// Numeric enums auto-increment
enum Level {
  Low = 0,
  Medium = 1,
  High = 2
}

// Or let TypeScript auto-increment
enum Priority {
  Low,      // 0
  Medium,   // 1
  High      // 2
}

// You can also start at any number
enum HttpStatus {
  OK = 200,
  NotFound = 404,
  ServerError = 500
}
```

Reverse Mapping (Numeric Enums Only)

With numeric enums, you can look up the name from the value:

```
enum Status {
  Active = 0,
  Inactive = 1
}

// Forward mapping: Status.Active → 0
console.log(Status.Active);  // 0
// Reverse mapping: 0 → "Active"
console.log(Status[0]);    // "Active"
const code: number = 0;
console.log(Status[code]);  // "Active"
```

⚡ Const Enums

Const enums are optimized for performance. TypeScript inlines the values at compile-time, so the enum object never exists at runtime.

```
const enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT"
}

function move(direction: Direction) {
  console.log(direction);
}

move(Direction.Up);

// Compiled JavaScript:
// function move(direction) { console.log(direction); }
// move("UP");
// Notice: Direction enum disappears! The value "UP" is inlined.
```

✅ **Key Benefit of Const Enums:** Smaller JavaScript bundle size. The enum object is eliminated during compilation.

💎 Literal Types

A **literal type** is a type that is exactly one specific value. TypeScript has literal types for strings, numbers, and booleans.

String Literal Types

```
type Size = "small" | "medium" | "large";

function setSize(size: Size) {
  console.log(`Size: ${size}`);
}

setSize("small");   // ✅
setSize("large");   // ✅
setSize("huge");    // ❌ Error! Not in the union
```

Number Literal Types

```
type DiceRoll = 1 | 2 | 3 | 4 | 5 | 6;

function rollDice(roll: DiceRoll) {
  console.log(`You rolled: ${roll}`);
}

rollDice(3);   // ✅
rollDice(7);   // ❌ Error! 7 is not 1-6
```

Boolean Literal Types

```
type OnOrOff = true | false;

// This is actually just boolean, but you can be more specific:
type Enabled = true;  // Only true is allowed
function enable(status: Enabled) {
  // status can only be true
}

enable(true);  // ✅
enable(false); // ❌ Error!
```

🤔 Enums vs Literal Types

Both can represent restricted values. Which should you use?

Feature	Enum	Literal Union
Syntax	enum Direction { Up, Down }	type Direction = "Up" | "Down"
Values	Can be numbers or strings	Usually strings (but can be numbers)
Reverse Mapping	Yes (for numbers)	No
Const Enum Optimization	Yes, zero runtime overhead	Already zero overhead
When to Use	Legacy code, reverse mapping needed	Modern code, most new projects

✅ **Modern Best Practice:** Use literal types (string unions) instead of enums. They're simpler and produce the same JavaScript.

🌎 Real-World Patterns

HTTP Status Codes

```
// Using const enum
const enum HttpStatus {
  OK = 200,
  Created = 201,
  BadRequest = 400,
  Unauthorized = 401,
  NotFound = 404,
  ServerError = 500
}

function handleResponse(status: HttpStatus) {
  switch (status) {
    case HttpStatus.OK:
      console.log("Success");
      break;
    case HttpStatus.NotFound:
      console.log("Not found");
      break;
  }
}
```

Event Types with Literal Union

```
type Event =
  | { type: "click"; x: number; y: number }
  | { type: "scroll"; delta: number }
  | { type: "keypress"; key: string };

function handleEvent(event: Event) {
  switch (event.type) {
    case "click":
      console.log(`Clicked at ${event.x}, ${event.y}`);
      break;
    case "scroll":
      console.log(`Scrolled ${event.delta}px`);
      break;
    case "keypress":
      console.log(`Pressed ${event.key}`);
      break;
  }
}
```

💻 Coding Challenges

Challenge 1: String Enum

Create a `Color` enum with at least 5 colors (Red, Green, Blue, Yellow, Purple). Write a function that accepts a color and returns a CSS color code.

- Define the enum with string values
- Create a function that maps colors to hex codes
- Test with all colors

Goal: Practice string enums and mapping to values.

→ Solution

Challenge 2: Literal Type Union

Create a `LogLevel` type using literal union that represents log levels: "debug", "info", "warn", "error". Write a logger function that outputs with appropriate formatting.

- Define literal union type
- Create logger with different prefixes per level
- Demonstrate exhaustiveness checking

Goal: Practice literal types and discriminated unions.

→ Solution

Challenge 3: Const Enum Optimization

Create a `UserRole` const enum with Admin, User, Guest. Build a permission checker that uses the const enum. Compare compile output with regular enum.

- Define const enum
- Create hasPermission function
- Understand compile-time inlining

Goal: Understand const enum optimization and when to use it.

→ Solution

📋 Best Practices

💡 Use Literal Unions Over Enums

Modern TypeScript code prefers literal unions (string | unions). They're simpler, produce the same output, and work better with type inference.

```
// ✅ Better
type Status = "active" | "inactive" | "pending";

// ❌ Legacy
enum Status {
 Active = "active",
 Inactive = "inactive",
 Pending = "pending"
}
```

⚡ Use Const Enums for Optimization

If you must use enums, use `const enum` for zero runtime overhead. The values are inlined during compilation.

🎯 What's Next

We've mastered constants and restricted types. Chapter 10 covers **Modules & Namespaces** — how to organize code into reusable, maintainable modules and manage scope effectively.