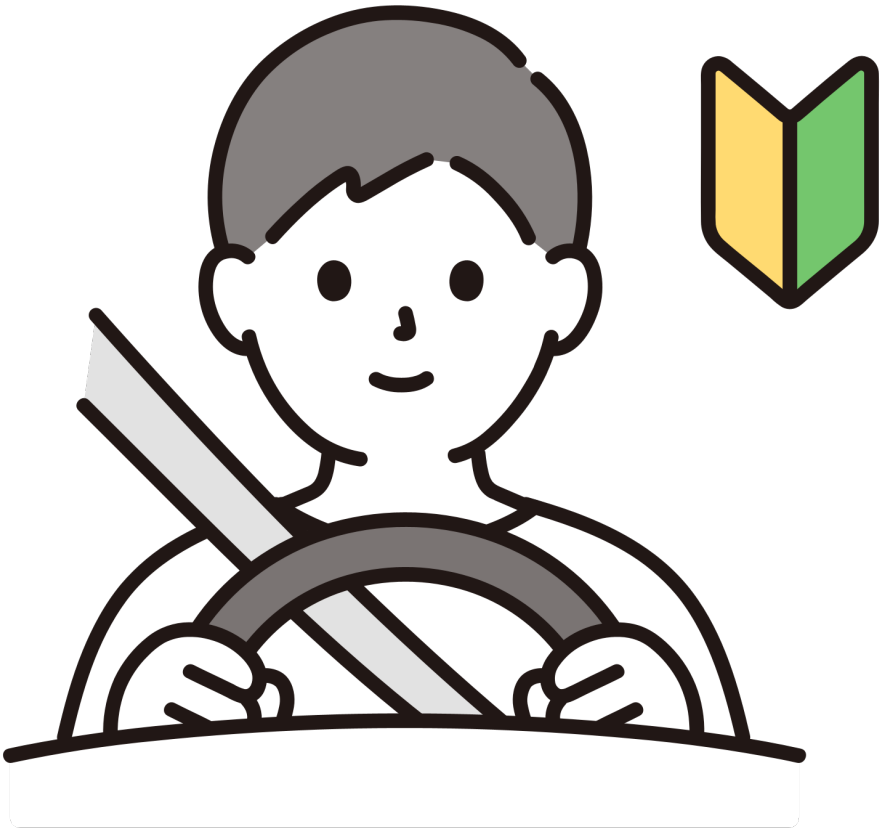
Typescript for Beginners



readbytes

Typescript for Beginners

From Novice to Advanced Programmer

readbytes.github.io

2025-07-28

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to TypeScript

1. What is TypeScript?
2. TypeScript vs JavaScript
3. Installing and Setting Up TypeScript
4. First TypeScript Program (`tsc`, `ts-node`)

1 Introduction to TypeScript

1.1 What is TypeScript?

TypeScript is a modern programming language developed by Microsoft that builds on top of JavaScript. You can think of it as **JavaScript with superpowers** — it adds new features that make writing and maintaining large, complex applications easier and safer.

At its core, TypeScript is a **superset of JavaScript**. This means every valid JavaScript program is also valid TypeScript. But TypeScript introduces an additional powerful feature: **static typing**. This allows you to specify the types of variables, function parameters, and return values upfront. This small addition brings huge benefits for developers, especially when working on big projects or with teams.

1.1.1 Why Was TypeScript Created?

JavaScript is incredibly flexible and ubiquitous, but it was originally designed for small scripts in web browsers. As applications grew larger and more complex, developers faced challenges like:

- Difficulty tracking down bugs caused by type errors (e.g., using a string where a number is expected).
- Lack of clear contracts between parts of the code.
- Reduced confidence when refactoring or extending code.
- Limited support from editors for autocompletion, error checking, and navigation.

TypeScript addresses these issues by introducing static types and a rich type system, enabling tools to catch many errors **before you run your code**.

1.1.2 The Role of TypeScript in Large-Scale Development

Because TypeScript adds type annotations and other features (like interfaces, enums, and generics), it helps you:

- **Catch errors early:** Many bugs can be detected while writing code, not just at runtime.
- **Write clearer code:** Types serve as documentation, making it easier for others (and future you) to understand what a piece of code expects and returns.
- **Improve tooling:** Editors can provide smarter autocompletion, refactoring tools, and jump-to-definition features.
- **Maintain codebases:** Large projects become more manageable as the type system enforces consistency and prevents accidental mistakes.

1.1.3 Simple Example: JavaScript vs TypeScript

Let's look at a very basic example to illustrate how TypeScript enhances JavaScript.

JavaScript (No Types)

Full runnable code:

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

console.log(add(5, 10));   // 15
console.log(add("5", 10)); // "510"
```

Here, the function `add` can accept any types for `a` and `b`. When we pass a string `"5"` and a number `10`, JavaScript concatenates them instead of adding numbers, which can lead to unexpected bugs.

TypeScript (With Types)

Full runnable code:

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

console.log(add(5, 10));    // 15
console.log(add("5", 10));  // Error: Argument of type 'string' is not assignable to parameter of type
```

In TypeScript, by declaring that both `a` and `b` must be numbers, the compiler immediately flags the incorrect call `add("5", 10)` as an error. This helps prevent bugs early in development.

1.1.4 Summary

TypeScript is a powerful extension of JavaScript that:

- Adds **static typing** and advanced language features.
- Improves **code quality**, **readability**, and **tooling support**.
- Is designed to handle the complexity of **large-scale application development**.
- Lets you write safer and more maintainable code while still being compatible with all existing JavaScript code.

In the next sections, we will explore how TypeScript compares directly to JavaScript, how to set up your environment, and write your very first TypeScript program.

1.2 TypeScript vs JavaScript

TypeScript and JavaScript share a close relationship — in fact, TypeScript is a **superset of JavaScript**, which means every JavaScript program is also valid TypeScript. However, TypeScript extends JavaScript with additional features that help developers write safer and more maintainable code.

In this section, we'll compare the two languages side-by-side to understand their **similarities** and **key differences**, focusing on syntax, type annotations, tooling, and the compilation process.

1.2.1 Similarities Between TypeScript and JavaScript

- **Same Core Syntax:** The basic syntax and language constructs (variables, functions, loops, conditionals, etc.) are identical. This makes it easy for JavaScript developers to start writing TypeScript code.
- **Runtime Behavior:** TypeScript compiles down to plain JavaScript. This means your TypeScript code, after compilation, behaves exactly like JavaScript when run in browsers or Node.js.
- **Compatibility:** You can gradually adopt TypeScript in your existing JavaScript projects by renaming `.js` files to `.ts` and adding type annotations incrementally.

1.2.2 Key Differences Between TypeScript and JavaScript

Aspect	JavaScript	TypeScript
Typing	Dynamically typed — types checked at runtime	Statically typed — types checked at compile time
Syntax	No type annotations	Supports type annotations (`: number`, `: string`, etc.)
Tooling	Basic editor support	Rich editor support with autocomplete, type checking, and refactoring tools
Compilation	Interpreted directly by runtime	Compiled (transpiled) to JavaScript before running
Advanced Features	ES features only (depending on environment)	Interfaces, enums, generics, namespaces, and more

1.2.3 Parallel Code Examples: JavaScript vs TypeScript

Example 1: Function without types

JavaScript

Full runnable code:

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

console.log(greet("Alice"));  // Hello, Alice
console.log(greet(42));       // Hello, 42 (unexpected)
```

TypeScript

Full runnable code:

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

console.log(greet("Alice"));  // Hello, Alice
console.log(greet(42));       // Error: Argument of type 'number' is not assignable to parameter of typ
```

Explanation: In JavaScript, you can pass any type to `greet`, even a number, which may cause unexpected output. TypeScript requires `name` to be a `string`, catching mistakes before running the code.

Example 2: Variable declarations with types

JavaScript

```javascript
let count = 10;
count = "ten";  // Allowed, but can cause bugs
```

TypeScript

```typescript
let count: number = 10;
count = "ten";  // Error: Type 'string' is not assignable to type 'number'.
```

Explanation: JavaScript variables are dynamically typed and can hold any type at any time. TypeScript variables can have explicit types, enforcing consistent usage.

Example 3: Using Interfaces (TypeScript only)

TypeScript introduces interfaces to define the shape of objects.

Full runnable code:

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

```ts
function printUser(user: User) {
  console.log(`${user.name} is ${user.age} years old.`);
}

printUser({ name: "Bob", age: 30 }); // Valid
printUser({ name: "Alice" });         // Error: Property 'age' is missing
```

Explanation: JavaScript does not have a native way to enforce object structure. TypeScript's interfaces add this capability, improving reliability.

1.2.4 Tooling and Compilation Differences

JavaScript

- Run directly by browsers or Node.js.
- Editor support depends on the IDE, usually basic.
- No compile step, errors show only at runtime.

TypeScript

- Needs to be **compiled** to JavaScript (`.ts` → `.js`) using the TypeScript compiler (`tsc`).
- Rich editor tooling with real-time error detection, autocomplete, and refactoring tools.
- Enables safer code by catching many common mistakes during compilation.

1.2.5 Summary

Feature	JavaScript	TypeScript
Runtime	Interpreted	Compiled to JavaScript
Typing	Dynamic	Static and optional
Syntax	Basic	Adds type annotations & advanced features
Tooling	Basic editor support	Enhanced IDE support
Safety	Errors detected at runtime	Errors detected at compile time

In essence, **TypeScript enhances JavaScript with types and tooling without changing its runtime behavior**. This means you get all the benefits of JavaScript's flexibility while adding safeguards that help build robust applications.

In the next section, we'll dive into how to install TypeScript and set up your development environment so you can start writing your own TypeScript code.

1.3 Installing and Setting Up TypeScript

In this section, we'll guide you step-by-step through setting up TypeScript on your local machine. By the end, you'll have a working TypeScript environment ready for writing and compiling TypeScript code.

1.3.1 Step 1: Install Node.js

TypeScript runs on top of Node.js, so the first step is to install Node.js if you don't have it already.

- Go to the official Node.js website: https://nodejs.org/
- Download the **LTS (Long Term Support)** version recommended for most users.
- Run the installer and follow the prompts to complete the installation.

Verify Node.js Installation

Open your terminal or command prompt and run:

```
node -v
```

You should see the version number, for example:

```
v20.1.0
```

1.3.2 Step 2: Install TypeScript Globally

With Node.js installed, you now have access to `npm` (Node Package Manager), which you will use to install TypeScript.

Run the following command to install the TypeScript compiler globally:

```
npm install -g typescript
```

The `-g` flag installs TypeScript globally so that you can use the `tsc` command anywhere on your system.

1.3.3 Step 3: Verify TypeScript Installation

After the installation completes, verify it by checking the TypeScript compiler version:

```
tsc -v
```

You should see output like this:

```
Version 5.1.3
```

This confirms that TypeScript is installed and ready to use.

1.3.4 Step 4: Create a Basic Project Structure

Let's create a folder for your TypeScript project.

1. Open your terminal and run:

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

2. Initialize a new Node.js project (optional but recommended):

```
npm init -y
```

This will create a `package.json` file, which helps manage project dependencies.

3. Create a folder for your TypeScript source files:

```
mkdir src
```

4. Inside the `src` folder, create a new file named `index.ts`. This will be your first TypeScript file.

1.3.5 Step 5: Create a Minimal `tsconfig.json`

TypeScript uses a configuration file called `tsconfig.json` to specify compiler options and which files to include.

To generate a basic `tsconfig.json` file, run:

```
tsc --init
```

This creates a file with many default settings. For now, the default configuration is enough for us to start.

Here is a minimal example of what `tsconfig.json` might look like:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true
  }
}
```

- **target**: The JavaScript version the TypeScript will compile down to (ES6 is modern

and widely supported).
- **module**: The module system used (`commonjs` is standard for Node.js).
- **outDir**: Output directory for compiled JavaScript files.
- **rootDir**: The folder where your TypeScript source files live.
- **strict**: Enables strict type-checking options for better code quality.

1.3.6 Step 6: Compile Your TypeScript Code

Now, add some simple code to `src/index.ts`:

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

To compile the TypeScript code into JavaScript, run:

```
tsc
```

This will read your `tsconfig.json` and compile all `.ts` files inside `src`, placing the JavaScript output inside the `dist` folder.

Check the `dist/index.js` file, which should look like this:

```
"use strict";
const greeting = "Hello, TypeScript!";
console.log(greeting);
```

1.3.7 Step 7: Run the Compiled JavaScript

Run the compiled JavaScript file using Node.js:

```
node dist/index.js
```

You should see this output in your terminal:

```
Hello, TypeScript!
```

1.3.8 Summary

Step	Command/Action
Install Node.js	Download from https://nodejs.org/
Install TypeScript	`npm install -g typescript`
Verify TypeScript	`tsc -v`

Step	Command/Action
Create Project Folder	`mkdir my-typescript-project && cd my-typescript-project`
Initialize Project	`npm init -y`
Create Source Folder	`mkdir src`
Initialize tsconfig	`tsc --init`
Compile Code	`tsc`
Run JavaScript Output	`node dist/index.js`

1.4 First TypeScript Program (`tsc`, `ts-node`)

Now that you have TypeScript installed and your project set up, it's time to write your **first TypeScript program**, compile it, and run it. In this section, we will walk through:

- Writing a simple TypeScript script
- Compiling it with the TypeScript compiler (`tsc`)
- Running the compiled JavaScript using Node.js
- Using `ts-node` for quick execution without manual compilation

1.4.1 Step 1: Write Your First TypeScript Script

Create a file named `hello.ts` inside your `src` folder. Open it in your code editor and add the following code:

Full runnable code:

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

This script declares a variable `greeting` with a type annotation `string`, assigns it a message, and prints it to the console.

1.4.2 Step 2: Compile the TypeScript File with `tsc`

To convert your TypeScript code into JavaScript that the runtime can execute, you need to compile it.

Open your terminal, navigate to your project directory, and run:

```
tsc src/hello.ts
```

This command runs the TypeScript compiler (`tsc`) on your `hello.ts` file and generates a JavaScript file with the same name but a `.js` extension in the **same folder** (`src/hello.js`).

1.4.3 Step 3: Understand the Output Files

- **Source file (`hello.ts`)**: Your original TypeScript code, which includes types and modern syntax.
- **Compiled file (`hello.js`)**: The plain JavaScript file generated by the compiler, free of any type annotations, ready to be run by Node.js or browsers.

You can open the generated `hello.js` file to see the compiled JavaScript:

Full runnable code:

```
"use strict";
const greeting = "Hello, TypeScript!";
console.log(greeting);
```

1.4.4 Step 4: Run the Compiled JavaScript Using Node.js

Now run the compiled JavaScript file with Node.js:

```
node src/hello.js
```

The output will be:

```
Hello, TypeScript!
```

1.4.5 Step 5: Using `ts-node` for Quick Execution (Without Manual Compilation)

Instead of compiling with `tsc` and then running the JavaScript output, you can use the tool `ts-node` to execute TypeScript files directly.

Install `ts-node`

If you haven't installed it yet, install it globally using npm:

```
npm install -g ts-node
```

Run the TypeScript File Directly

Now you can run your `hello.ts` script without manually compiling:

```
ts-node src/hello.ts
```

This will output:

```
Hello, TypeScript!
```

How does this work? `ts-node` compiles your TypeScript code in memory and immediately runs it, skipping the step of generating a JavaScript file on disk. This is very handy for quick testing and development.

1.4.6 Summary: When to Use `tsc` vs `ts-node`

Task	Command	Notes
Compile TypeScript to JavaScript	`tsc src/hello.ts`	Generates `.js` files you can run later
Run compiled JavaScript	`node src/hello.js`	Runs the compiled JavaScript
Run TypeScript directly	`ts-node src/hello.ts`	Compiles & runs in one step; no `.js` files created

1.4.7 Recap

- Your `.ts` files contain TypeScript code with type annotations.
- The TypeScript compiler (`tsc`) converts `.ts` files to `.js` files.
- Node.js runs the compiled `.js` files.
- `ts-node` allows running `.ts` files directly without manual compilation.

With this workflow in place, you're now ready to write, compile, and run TypeScript programs! In the next chapters, we will dive deeper into TypeScript's type system and features to help you write safer and more powerful code.

Chapter 2.

Type Annotations and Basic Types

1. Number, String, Boolean
2. Arrays and Tuples
3. `any`, `unknown`, `void`, `null`, `undefined`, and `never`
4. Type Inference

2 Type Annotations and Basic Types

2.1 Number, String, Boolean

In TypeScript, understanding and using **basic types** is essential to writing clear and reliable code. The three most fundamental types you'll encounter are:

- `number`
- `string`
- `boolean`

These types let you specify exactly what kind of values a variable can hold, helping catch errors early and making your code easier to understand.

2.1.1 Number

The `number` type represents both integers and floating-point numbers.

Declaring a `number` variable:

```
let age: number = 30;
let price: number = 19.99;
```

Operations with numbers:

Full runnable code:

```
let x: number = 10;
let y: number = 5;
let sum: number = x + y;
console.log(sum);  // Output: 15
```

Type Error Example:

If you try to assign a non-number value, TypeScript will show an error:

```
let score: number = 100;
score = "high"; // Error: Type 'string' is not assignable to type 'number'.
```

2.1.2 String

The `string` type represents text values.

Declaring a `string` variable:

```
let name: string = "Alice";
let greeting: string = 'Hello, world!';
```

String operations:

Full runnable code:

```
let firstName: string = "John";
let lastName: string = "Doe";
let fullName: string = firstName + " " + lastName;
console.log(fullName);  // Output: John Doe
```

Type Error Example:

```
let message: string = "Welcome!";
message = 123;  // Error: Type 'number' is not assignable to type 'string'.
```

Template Literals and Type-Safe Concatenation

TypeScript fully supports **template literals**, which make string concatenation cleaner and easier to read:

Full runnable code:

```
let user: string = "Jane";
let age: number = 28;
let info: string = `${user} is ${age} years old.`;
console.log(info);  // Output: Jane is 28 years old.
```

Using template literals helps keep types consistent because the expressions inside `${}` must respect their declared types.

2.1.3 Boolean

The `boolean` type represents logical values: `true` or `false`.

Declaring a `boolean` variable:

```
let isOnline: boolean = true;
let hasAccess: boolean = false;
```

Using booleans in conditions:

Full runnable code:

```
let loggedIn: boolean = true;
```

```
if (loggedIn) {
  console.log("User is logged in.");
} else {
  console.log("User is not logged in.");
}
```

Type Error Example:

```
let isAvailable: boolean = true;
isAvailable = "yes";  // Error: Type 'string' is not assignable to type 'boolean'.
```

2.1.4 Summary

Type	Example Declaration	Allowed Values	Error Example
`number`	`let age: number = 30;`	`10`, `3.14`, `-100`	`age = "thirty";` (Error)
`string`	`let name: string = "Bob";`	`"hello"`, `'world'`	`name = 123;` (Error)
`boolean`	`let flag: boolean = true;`	`true`, `false`	`flag = "true";` (Error)

Mastering these basic types is the first step toward writing clear, type-safe code in TypeScript. In the next section, we'll explore how to work with collections of values using **arrays** and **tuples**.

2.2 Arrays and Tuples

In TypeScript, arrays and tuples allow you to work with collections of values. They provide a way to store multiple items and give you tools to enforce type safety when accessing or modifying those collections.

2.2.1 Arrays

Arrays are lists of elements where all items typically share the same type.

Declaring Arrays

You can declare an array in two equivalent ways:

1. **Using `type[]` syntax:**

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

2. **Using `Array<type>` syntax:**

```
let numbers: Array<number> = [5, 6, 7, 8];
let fruits: Array<string> = ["mango", "pineapple", "grape"];
```

Both styles work the same, and you can use whichever you prefer.

Accessing and Modifying Arrays

You can access elements using their index (starting at 0) and modify them safely thanks to type annotations:

Full runnable code:

```
let colors: string[] = ["red", "green", "blue"];
console.log(colors[0]);  // Output: red

colors[1] = "yellow";    // Change 'green' to 'yellow'
console.log(colors);     // Output: ['red', 'yellow', 'blue']
```

Attempting to assign a value of a different type will cause an error:

```
colors[2] = 10;  // Error: Type 'number' is not assignable to type 'string'.
```

Looping Over Arrays

TypeScript fully supports JavaScript's array methods and loops, with type safety intact:

Full runnable code:

```
let scores: number[] = [85, 92, 78, 99];

// Using for...of loop
for (const score of scores) {
  console.log(score);
}

// Using forEach method
scores.forEach((score) => {
  console.log(score * 2);  // Multiply each score by 2
});
```

2.2.2 Tuples

Tuples are special fixed-length arrays where each element has a known type, and the order of types matters.

Declaring Tuples

Tuples are useful when you want to group different types together in a fixed structure:

```
let coordinate: [number, number] = [10, 20];
let person: [string, number] = ["Alice", 30];
```

- The first element in `coordinate` must be a number (e.g., x-coordinate).
- The second element must be a number (e.g., y-coordinate).
- The `person` tuple holds a string (name) and a number (age).

Accessing Tuple Elements

You can access tuple elements by their index:

```
console.log(coordinate[0]);  // Output: 10
console.log(person[1]);      // Output: 30
```

Trying to assign the wrong type or add extra elements will cause an error:

```
coordinate[0] = "left";   // Error: Type 'string' is not assignable to type 'number'.
coordinate.push(50);      // Error: Tuple type '[number, number]' of length '2' has no element at index
```

Use Case: Function Returning a Tuple

Tuples are great for functions that return multiple values of different types:

Full runnable code:

```
function getUserInfo(): [string, number] {
  return ["Bob", 25];
}

const userInfo = getUserInfo();
console.log(`Name: ${userInfo[0]}, Age: ${userInfo[1]}`);
// Output: Name: Bob, Age: 25
```

2.2.3 Summary

Feature	Syntax Examples	Description
Array	`let nums: number[] = [1, 2];`	List of values with the same type
Array	`let fruits: Array<string> = [...]`	Generic array syntax equivalent
Tuple	`let point: [number, number]`	Fixed-length array with fixed types
Tuple Example	`let person: [string, number]`	Structured data with different types

Arrays and tuples are powerful ways to organize data with TypeScript's type safety. Arrays

are perfect when working with collections of the same type, while tuples shine when grouping a fixed number of elements of different types.

Next, we'll explore some special types like `any`, `unknown`, and others that help in more complex scenarios.

2.3 `any`, `unknown`, `void`, `null`, `undefined`, and `never`

TypeScript includes several **special types** designed to handle particular programming scenarios, including uncertainty, absence of values, and functions with no meaningful return. Understanding these types helps you write safer and more expressive code.

2.3.1 `any`

Purpose

The `any` type disables type checking for a variable, allowing it to hold **any value** without compiler errors.

Example

```
let data: any = 42;
data = "Hello";
data = true;
```

When to Use or Avoid

- Use `any` when migrating JavaScript code to TypeScript gradually, or when you really cannot know the variable's type.
- **Avoid** using `any` in general because it removes the benefits of type safety and defeats the purpose of TypeScript.

Warning

```
let value: any = 10;
value.foo(); // No error, but will cause runtime error if foo() doesn't exist
```

Here, `any` lets you call any property or method, but this can lead to runtime bugs.

2.3.2 `unknown`

Purpose

`unknown` is a safer alternative to `any`. It allows any value but **forces you to check the type before using it**.

Example

Full runnable code:

```
let input: unknown = 5;
input = "hello";

if (typeof input === "string") {
  console.log(input.toUpperCase()); // Safe: TypeScript knows input is string here
}
```

When to Use

- Use `unknown` when the type of a variable is not known upfront but you want to maintain type safety.
- It requires explicit type checks or assertions before use.

2.3.3 Comparing `any` vs `unknown`

Feature	`any`	`unknown`
Accepts any value	YES	YES
Allows any operation	YES (unsafe)	NO (requires type checking)
Type safety	None	Enforced

2.3.4 `void`

Purpose

`void` is used to indicate that a function **does not return any value**.

Example

```
function logMessage(message: string): void {
  console.log(message);
}
```

Here, the function returns nothing (`undefined` implicitly), so we annotate its return type as

`void`.

When to Use

- Use `void` as the return type of functions that perform actions but don't return data.
- Avoid assigning `void` to variables (except `undefined`).

2.3.5 `null` and `undefined`

Purpose

- `null` represents an **explicit absence of any value**.
- `undefined` means a variable has **been declared but not assigned a value**.

Example

```
let n: null = null;
let u: undefined = undefined;

let name: string | null = null;   // Name can be string or null
let age: number | undefined;       // Age can be number or undefined
```

Notes

- By default, `null` and `undefined` are **not assignable** to other types unless explicitly allowed.
- They are commonly used to model optional or missing data.

2.3.6 `never`

Purpose

`never` represents values that **never occur**. It is mainly used to describe:

- Functions that **never return** (e.g., throw errors or infinite loops).
- Impossible code paths.

Example

```
function error(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {}
}
```

When to Use

- Use `never` as a return type for functions that don't complete normally.
- Helps the compiler identify unreachable code.

2.3.7 Summary Table

Type	Purpose	Usage Example	When to Use / Avoid
`any`	Disable type checking, hold any value	`let x: any = 10; x = "hello";`	Use sparingly; avoid if possible
`unknown`	Hold any value but require type checks	`let x: unknown = 10; if (typeof x === "number") {...}`	Safer alternative to `any`
`void`	Function returns nothing	`function log(): void {}`	For functions with no return
`null`	Explicit absence of value	`let n: null = null;`	Model missing values
`undefined`	Variable declared but uninitialized	`let u: undefined = undefined;`	Model missing or optional values
`never`	Represents no possible value (e.g., throws)	`function fail(): never { throw new Error(); }`	Functions that never return

2.3.8 Conclusion

These special types help TypeScript model real-world programming situations more precisely:

- Use `unknown` over `any` to keep type safety.
- Use `void` for functions without return values.
- Use `null` and `undefined` to represent absent or optional values explicitly.
- Use `never` to signal unreachable code or non-returning functions.

Understanding these types strengthens your ability to write robust and predictable TypeScript code. Next, we will explore how TypeScript infers types automatically, reducing the need for explicit annotations.

2.4 Type Inference

One of TypeScript's powerful features is **type inference** — the ability to automatically deduce the type of a variable or expression based on the value you assign to it. This means you don't always have to explicitly specify types, which helps keep your code concise without losing type safety.

2.4.1 How Type Inference Works

When you declare and initialize a variable, TypeScript looks at the assigned value and **infers the most specific type** it can.

Example: Inferred Types for Variables

```
let count = 42;         // TypeScript infers count is of type 'number'
let message = "Hi!";    // message is inferred as 'string'
let isReady = true;     // isReady is inferred as 'boolean'
```

If you try to assign a different type later, TypeScript will raise an error:

```
count = 100;            // OK
count = "one hundred"; // Error: Type 'string' is not assignable to type 'number'
```

2.4.2 Type Inference in Functions

TypeScript can also infer types for function return values based on what you return inside the function.

```
function multiply(a: number, b: number) {
  return a * b;  // Return type inferred as 'number'
}

let result = multiply(2, 3);  // result inferred as 'number'
```

Here, even though the return type is not explicitly declared, TypeScript understands it from the return statement.

2.4.3 Benefits of Type Inference

- **Less boilerplate:** You don't need to write types everywhere, making code cleaner.
- **Type safety:** TypeScript still enforces type correctness based on what it infers.
- **Improved developer experience:** Autocomplete and error checking work with inferred types.

- **Faster development:** You can write code quickly while still getting strong type checking.

2.4.4 When to Use Explicit Type Annotations

Sometimes, type inference is not enough or can lead to unintended results. In these cases, adding explicit type annotations improves clarity and prevents subtle bugs.

Example 1: Variables initialized with `null` or `undefined`

```
let data = null;  // Type inferred as 'null', which is usually not what you want

data = "Hello";   // Error: Type 'string' is not assignable to type 'null'
```

Solution:

```
let data: string | null = null;  // Explicitly say 'data' can be string or null
data = "Hello";                  // OK now
```

Example 2: Functions with no return statement or multiple return types

```
function getStatus(flag: boolean) {
  if (flag) {
    return "success";
  }
  // No return in else case - inferred return type is 'string | undefined'
}
```

To make the return type clear and avoid bugs, add an explicit annotation:

```
function getStatus(flag: boolean): string {
  if (flag) {
    return "success";
  }
  return "failure"; // ensure all paths return a string
}
```

2.4.5 Summary

Scenario	Inference Behavior	Recommendation
Variable initialized with value	Type inferred from the value	Usually no annotation needed
Variable initialized with `null` or `undefined`	Inferred type can be narrow (e.g., `null`)	Add explicit union type (e.g., ‘string | null‘)
Function return type	Inferred from returned expressions	Add explicit annotation if ambiguous

Scenario	Inference Behavior	Recommendation
Complex types or intent unclear	May infer overly broad or narrow types	Use explicit type annotations

2.4.6 Conclusion

TypeScript's type inference helps you write less verbose code without sacrificing safety. It automatically determines types from your assignments and function returns, providing helpful tooling and error detection.

However, knowing **when to add explicit type annotations** is key to maintaining clarity and preventing unexpected bugs. Use inference for simple cases and annotations when your intent or type is not obvious.

Chapter 3.

Working with Variables and Constants

1. `let`, `const`, and `var`
2. Scope and Shadowing
3. Best Practices for Declarations

3 Working with Variables and Constants

3.1 `let`, `const`, and `var`

When declaring variables in TypeScript (and JavaScript), you have three keywords available: `let`, `const`, and `var`. Understanding their differences is important for writing clear, predictable, and bug-free code.

3.1.1 `var`

- **Scope:** Function-scoped or global-scoped.
- **Hoisting:** Variables declared with `var` are hoisted (moved to the top of their function or global scope), which can lead to unexpected behavior.
- **Reassignable:** You can reassign and redeclare variables declared with `var`.

Example: `var` and function scope

Full runnable code:

```
function example() {
  if (true) {
    var message = "Hello from var";
  }
  console.log(message); // Output: Hello from var
}

example();
```

Even though `message` is declared inside the `if` block, it is accessible outside that block within the function because `var` is function-scoped.

Pitfall: `var` hoisting

Full runnable code:

```
console.log(num);  // Output: undefined (due to hoisting)
var num = 5;
```

`num` is hoisted and initialized as `undefined` before assignment, which can cause confusion.

3.1.2 `let`

- **Scope:** Block-scoped (only accessible within the nearest enclosing `{ }`).
- **Hoisting:** Variables declared with `let` are hoisted but not initialized, so accessing

them before declaration causes an error.
- **Reassignable:** You can reassign but **cannot redeclare** the variable in the same scope.

Example: `let` and block scope

Full runnable code:

```
if (true) {
  let greeting = "Hello from let";
  console.log(greeting); // Output: Hello from let
}
// console.log(greeting); // Error: Cannot find name 'greeting'
```

`greeting` exists only inside the `if` block and is not accessible outside.

3.1.3 `const`

- **Scope:** Block-scoped like `let`.
- **Hoisting:** Same as `let`.
- **Immutable binding:** Variables declared with `const` **cannot be reassigned** after initialization.
- Must be initialized at declaration.

Example: `const` and immutability

Full runnable code:

```
const PI = 3.14;
console.log(PI);  // Output: 3.14

// PI = 3.1415;  // Error: Cannot assign to 'PI' because it is a constant
```

Important Note About `const` and Objects/Arrays

`const` prevents reassigning the variable itself but **does not make objects or arrays immutable**:

Full runnable code:

```
const user = { name: "Alice" };
user.name = "Bob";  // Allowed: properties can be changed
console.log(JSON.stringify(user,null,2));

const numbers = [1, 2, 3];
numbers.push(4);    // Allowed: array contents can change
console.log(numbers);
```

3.1.4 Why Use `let` and `const` Instead of `var`?

Modern TypeScript (and JavaScript) development favors `let` and `const` because:

- **Block scoping** prevents unintended variable access outside of its logical block, reducing bugs.
- `const` clearly signals variables that should not be reassigned, improving code readability and intent.
- `let` avoids the pitfalls of `var`'s hoisting and redeclaration issues.
- Many style guides and linters enforce using `let` and `const` exclusively for safer, cleaner code.

3.1.5 Summary Table

Keyword	Scope	Reassignable?	Redeclarable?	Hoisting Behavior	Use Case
`var`	Function/global	Yes	Yes	Hoisted and initialized	Legacy code, avoid in new code
`let`	Block	Yes	No	Hoisted but not initialized	Variables that will change value
`const`	Block	No (binding only)	No	Hoisted but not initialized	Constants or values that don't reassign

3.1.6 Example: Putting It All Together

Full runnable code:

```
function demo() {
  if (true) {
    var x = 1;
    let y = 2;
    const z = 3;
  }

  console.log(x); // 1 (var is function-scoped)
  // console.log(y); // Error: y is not defined (let is block-scoped)
  // console.log(z); // Error: z is not defined (const is block-scoped)
}

demo();
```

By preferring `const` and `let` over `var`, you write code that is easier to understand, less

error-prone, and better aligned with modern best practices in TypeScript development. Use `const` whenever possible, and `let` only when a variable's value needs to change.

3.2 Scope and Shadowing

Understanding **variable scope** is crucial for managing where variables are accessible and how their values can be affected within different parts of your code. In TypeScript (and JavaScript), scope defines the visibility and lifetime of variables.

3.2.1 Types of Scope

Global Scope

Variables declared outside any function or block have **global scope**. They are accessible anywhere in the code.

Full runnable code:

```
let globalVar = "I'm global";

function printGlobal() {
  console.log(globalVar);  // Accessible here
}

printGlobal();  // Output: I'm global
console.log(globalVar);    // Accessible here too
```

Function Scope

Variables declared inside a function with `var`, `let`, or `const` are only accessible within that function.

Full runnable code:

```
function myFunction() {
  let funcVar = "I'm inside a function";
  console.log(funcVar);  // Accessible here
}

myFunction();
// console.log(funcVar);  // Error: funcVar is not defined outside the function
```

Block Scope

Variables declared with `let` or `const` inside blocks (anything inside `{}` such as `if`, `for`, or `{}` braces) are only accessible within that block.

Full runnable code:

```
if (true) {
  let blockVar = "I'm inside an if block";
  console.log(blockVar);  // Accessible here
}

// console.log(blockVar);  // Error: blockVar is not defined outside the block
```

Variables declared with `var` do **not** have block scope, only function or global scope.

3.2.2 Variable Shadowing

Shadowing occurs when a variable declared in an inner scope has the same name as a variable in an outer scope. The inner variable **"shadows"** the outer one, making the outer variable temporarily inaccessible inside the inner scope.

Example of Shadowing

Full runnable code:

```
let message = "Outer message";

function showMessage() {
  let message = "Inner message";  // Shadows outer 'message'
  console.log(message);            // Output: Inner message
}

showMessage();
console.log(message);              // Output: Outer message
```

Here, the inner `message` variable inside the function shadows the outer one. Outside the function, the outer variable is still accessible.

Shadowing in Nested Blocks

Shadowing can also happen inside nested blocks:

Full runnable code:

```
let count = 10;

if (true) {
  let count = 20;  // Shadows outer 'count' within this block
  console.log(count);  // Output: 20
}

console.log(count);      // Output: 10
```

Inside the `if` block, `count` refers to the inner variable, while outside, it refers to the outer

one.

3.2.3 Why Shadowing Can Be Problematic

- Shadowing can lead to **confusing bugs** because you might accidentally refer to or update the wrong variable.
- It makes code harder to read and maintain because the same name refers to different variables in different places.

3.2.4 How TypeScript Helps with Shadowing

TypeScript's compiler can **warn you about variable shadowing**, helping you catch potential issues before runtime.

For example, if you declare a variable in an inner scope with the same name as one in an outer scope, TypeScript can show a warning or error (depending on your configuration), prompting you to rename variables or rethink your design.

3.2.5 Summary

Scope Type	Declaration Keyword(s)	Scope Description
Global	`var`, `let`, `const`	Accessible everywhere
Function	`var`, `let`, `const`	Accessible only inside the function
Block	`let`, `const`	Accessible only inside the block `{}`
Block (var)	`var`	Does *not* have block scope; function or global scoped

3.2.6 Tips to Avoid Shadowing Bugs

- Use **meaningful and unique variable names**.
- Prefer `const` and `let` to limit scope and accidental redeclarations.
- Pay attention to compiler warnings about shadowing.
- Refactor code to minimize deep nesting and overlapping variable names.

By understanding scope and shadowing, you can better control variable visibility and avoid

subtle bugs that can arise when multiple variables share the same name in different scopes. TypeScript's static analysis helps by catching these issues early, leading to more maintainable code.

3.3 Best Practices for Declarations

Writing clean and predictable variable declarations is essential for maintainable and bug-free TypeScript code. This section provides practical guidelines to help you write declarations that are clear, efficient, and easy to understand.

3.3.1 Prefer `const` Over `let` Where Possible

Use `const` by default, and only use `let` if you need to reassign the variable later. This signals to readers of your code that the variable should not change, making your intent clear.

Poor Practice (overusing `let`)

```
let maxScore = 100;
let userName = "Alice";

maxScore = 120;  // Reassigning maxScore unnecessarily
```

Good Practice (use `const` when reassignment is not needed)

```
const maxScore = 100;
const userName = "Alice";
// maxScore = 120;  // Error: Cannot assign to 'maxScore' because it is a constant.
```

3.3.2 Avoid `var` Entirely

The `var` keyword is function-scoped and hoisted, which can lead to hard-to-debug errors. Modern TypeScript and JavaScript favor block-scoped `let` and `const` instead.

Poor Practice (`var` usage)

```
function demo() {
  if (true) {
    var temp = 5;
  }
  console.log(temp);  // Accessible outside block - unexpected!
}
```

Good Practice (`let` or `const` usage)

```
function demo() {
  if (true) {
    const temp = 5;
    console.log(temp);  // Accessible here
  }
  // console.log(temp);  // Error: 'temp' is not defined here
}
```

3.3.3 Use Meaningful Variable Names

Choose descriptive names that clearly convey the purpose or content of the variable. Avoid vague names like `a`, `temp`, or `data` unless in very limited, obvious contexts.

Poor Practice (unclear names)

```
let x = 10;
let y = 20;
let z = x + y;
```

Good Practice (meaningful names)

```
let applesCount = 10;
let orangesCount = 20;
let totalFruits = applesCount + orangesCount;
```

3.3.4 Declare Variables Close to Where Theyre Used

Declare variables as near as possible to where they're first needed. This limits their scope and makes the code easier to read and maintain.

Poor Practice (declare variables far from usage)

```
let result;

function calculate() {
  // many lines of code
  result = 42;  // assigned much later
  console.log(result);
}
```

Good Practice (declare variables close to use)

```
function calculate() {
  const result = 42;
  console.log(result);
```

```
}
```

3.3.5 Summary of Best Practices

Practice	Why It Matters
Prefer `const` over `let`	Signals immutability; prevents accidental reassignments
Avoid `var`	Prevents scope and hoisting-related bugs
Use meaningful variable names	Improves code readability and intent
Declare variables close to usage	Limits scope; makes code easier to understand

3.3.6 Final Example: Putting It All Together

```
function calculateTotalPrice(prices: number[]): number {
  const taxRate = 0.07;  // Tax rate does not change
  let subtotal = 0;      // Will be updated while summing prices

  for (const price of prices) {
    subtotal += price;
  }

  const totalPrice = subtotal + subtotal * taxRate;
  return totalPrice;
}
```

This example uses `const` for values that do not change, `let` for variables that update, descriptive names, and declares variables close to their use.

Following these best practices will help you write clear, maintainable, and bug-resistant TypeScript code that others (and your future self!) will appreciate.

Chapter 4.

Functions in TypeScript

1. Function Types and Return Types
2. Optional and Default Parameters
3. Rest Parameters
4. Arrow Functions

4 Functions in TypeScript

4.1 Function Types and Return Types

Functions are a core part of any programming language, and TypeScript enhances them by allowing you to **annotate parameter types and return types**. This helps catch errors early and makes your code easier to understand and maintain.

4.1.1 Declaring Functions with Type Annotations

You can specify the type of each parameter as well as the return type of the function.

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

- `a: number` and `b: number` are parameter type annotations.
- `: number` after the parameter list indicates the function returns a number.

4.1.2 Assigning Functions to Variables with Function Types

Functions can also be assigned to variables. When doing so, you can annotate the variable with a **function type** specifying parameter types and return type.

Full runnable code:

```
let multiply: (x: number, y: number) => number;

multiply = function(x, y) {
  return x * y;
};

console.log(multiply(3, 4));  // Output: 12
```

Here, `multiply` must be a function that takes two numbers and returns a number.

4.1.3 Specifying Return Types Explicitly

TypeScript often **infers** the return type by analyzing the function body, but sometimes explicitly specifying the return type is helpful:

When TypeScript infers return type:

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

Here, the return type is inferred as `string` from the template literal.

When to specify return type explicitly:

- When the function returns `void` (nothing)
- To improve readability or document intent
- To avoid accidental changes in return type
- When the function has complex logic or multiple return paths

```
function logMessage(message: string): void {
  console.log(message);
}
```

4.1.4 Function Expressions with Annotations

You can also annotate function expressions inline:

```
const divide = (a: number, b: number): number => {
  return a / b;
};
```

Or with type annotations on the variable:

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

4.1.5 Summary

Syntax	Description
`function fn(a: type): returnType {}`	Named function with parameter and return types
`let fnVar: (a: type) => returnType`	Variable with a function type annotation
`const fn = (a: type): returnType => {}`	Arrow function with inline annotations

4.1.6 Example: Putting It All Together

Full runnable code:

```
// Named function with annotations
function subtract(a: number, b: number): number {
  return a - b;
}

// Function assigned to a variable with function type
let calculate: (x: number, y: number) => number;

calculate = subtract;

console.log(calculate(10, 4));  // Output: 6

// Arrow function with inline annotations
const greet = (name: string): string => `Hello, ${name}!`;

console.log(greet("Alice"));    // Output: Hello, Alice!
```

Annotating function parameter and return types makes your TypeScript code safer and clearer, preventing bugs and improving developer experience through better tooling and autocomplete.

4.2 Optional and Default Parameters

TypeScript lets you make function parameters **optional** or provide **default values** for them. These features make your functions more flexible and easier to use in different scenarios.

4.2.1 Optional Parameters (?)

To declare a parameter as optional, add a `?` after its name. Optional parameters **must come after all required parameters** in the function signature.

Syntax

Full runnable code:

```
function greet(name: string, greeting?: string) {
  if (greeting) {
    console.log(`${greeting}, ${name}!`);
  } else {
    console.log(`Hello, ${name}!`);
  }
}
```

```
greet("Alice");                  // Output: Hello, Alice!
greet("Bob", "Good morning");    // Output: Good morning, Bob!
```

Here, the `greeting` parameter is optional. If it's not provided, the function uses a default greeting.

4.2.2 Default Parameters

You can provide a **default value** for parameters. If the caller omits that argument or passes `undefined`, the default value is used.

Syntax

Full runnable code:

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

greet("Alice");                  // Output: Hello, Alice!
greet("Bob", "Good morning");    // Output: Good morning, Bob!
```

The behavior looks similar to optional parameters but with subtle differences.

4.2.3 Optional Parameters vs Default Parameters

Feature	Optional Parameters (`?`)	Default Parameters (`=`)
Can be omitted	Yes	Yes
Default value	No; value is `undefined` if omitted	Yes; default value used if omitted or `undefined`
Can be `undefined`	Yes	Usually no (because default replaces it)
Function body handles	Usually needs explicit checks for `undefined`	No need to check because default applies
Parameter position	Must come after required parameters	Can appear anywhere (though often after required)

4.2.4 Important: Order of Parameters

Optional parameters **must come after** all required ones:

```
function badExample(optionalParam?: string, requiredParam: number) {
  // Error: A required parameter cannot follow an optional parameter
}
```

Correct way:

```
function goodExample(requiredParam: number, optionalParam?: string) {
  // OK
}
```

4.2.5 Examples Combining Both Concepts

Function with Optional Parameter and Conditional Logic

Full runnable code:

```
function sendMessage(to: string, message?: string) {
  if (message) {
    console.log(`Sending message to ${to}: ${message}`);
  } else {
    console.log(`Sending default message to ${to}`);
  }
}

sendMessage("Alice");                       // Uses default message
sendMessage("Bob", "Hi Bob, how are you?"); // Uses provided message
```

Function with Default Parameters

Full runnable code:

```
function multiply(a: number, b: number = 1): number {
  return a * b;
}

console.log(multiply(5));     // Output: 5 (b defaults to 1)
console.log(multiply(5, 2));  // Output: 10 (b is 2)
```

4.2.6 Summary

Parameter Type	Declaration Example	Behavior
Required	`param: type`	Must be provided when calling
Optional	`param?: type`	Can be omitted; value is `undefined` if omitted
Default	`param: type = defaultValue`	Can be omitted; uses default value if omitted

Using optional and default parameters wisely lets you write flexible and robust functions that handle a variety of input scenarios with ease. Next, we'll explore how to handle an arbitrary number of arguments using rest parameters.

4.3 Rest Parameters

In TypeScript, **rest parameters** allow you to accept a **variable number of arguments** in a function. This is useful when you don't know in advance how many arguments will be passed.

Rest parameters are declared using the `...` syntax and are always typed as arrays. TypeScript enforces the type of each element in that array.

4.3.1 Syntax of Rest Parameters

```
function functionName(...parameterName: type[]) {
  // function body
}
```

- `...parameterName` gathers all remaining arguments into an array.
- `type[]` defines the type of elements expected.

4.3.2 Example 1: Summing Any Number of Numbers

Full runnable code:

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

console.log(sum(1, 2, 3));          // Output: 6
console.log(sum(10, 20, 30, 40));   // Output: 100
```

```
console.log(sum());                 // Output: 0
```

- The rest parameter `...numbers` is typed as `number[]`.
- TypeScript ensures only numbers can be passed to this function.
- The `reduce` method adds all the numbers together.

4.3.3 Example 2: Joining Strings

Full runnable code:

```
function joinStrings(separator: string, ...parts: string[]): string {
  return parts.join(separator);
}

console.log(joinStrings(", ", "apple", "banana", "cherry"));
// Output: "apple, banana, cherry"

console.log(joinStrings(" - ", "a", "b", "c", "d"));
// Output: "a - b - c - d"
```

- The first parameter (`separator`) is a regular string.
- The rest parameter `...parts` captures all remaining arguments as a `string[]`.
- TypeScript enforces that only strings can be passed after the separator.

4.3.4 Type Enforcement with Rest Parameters

If you try to pass an argument of the wrong type, TypeScript will throw an error.

Full runnable code:

```
function logNumbers(...values: number[]) {
  values.forEach(v => console.log(v));
}

logNumbers(1, 2, 3);         // YES OK
// logNumbers(1, "two", 3); // NO Error: Argument of type 'string' is not assignable to parameter of ty
```

4.3.5 Notes

- Rest parameters **must be the last parameter** in the function.
- You can mix regular and rest parameters, as long as the rest parameter comes at the end.

Full runnable code:

```
function greetUsers(greeting: string, ...names: string[]) {
  for (const name of names) {
    console.log(`${greeting}, ${name}!`);
  }
}

greetUsers("Hello", "Alice", "Bob", "Charlie");
// Output:
// Hello, Alice!
// Hello, Bob!
// Hello, Charlie!
```

4.3.6 Summary

Feature	Behavior
`...args: type[]`	Accepts a variable number of arguments as an array
Type enforcement	All elements must match the specified type
Parameter position	Rest parameter must come last in the parameter list
Use cases	Aggregating values, flexible APIs, variadic functions

Using rest parameters in TypeScript allows you to write **flexible, reusable, and type-safe** functions that handle varying numbers of inputs without sacrificing clarity or safety.

Next, we'll look at how **arrow functions** provide a concise way to write functions with simpler syntax and predictable `this` behavior.

4.4 Arrow Functions

Arrow functions in TypeScript (and JavaScript) provide a **concise and modern** way to write function expressions. They are especially useful for short, one-line functions and are commonly used with array methods and callbacks.

Arrow functions also **preserve the `this` context** of the surrounding scope, which makes them behave differently from traditional functions in some cases (especially in classes or event handlers).

4.4.1 Basic Arrow Function Syntax

```
// Traditional function expression
const add = function(a: number, b: number): number {
  return a + b;
};

// Equivalent arrow function
const addArrow = (a: number, b: number): number => {
  return a + b;
};
```

4.4.2 Implicit vs Explicit Return

Arrow functions can **omit the `return` keyword and braces** when returning a single expression. This is called an **implicit return**.

Explicit Return

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

Implicit Return

```
const multiply = (x: number, y: number): number => x * y;
```

If the function body is a single expression, you can use implicit return to write more concise code.

4.4.3 Arrow Functions in Array Methods

Arrow functions are often used with array methods like `.map()`, `.filter()`, and `.reduce()` for clarity and brevity.

Example: `.map()` with arrow function

Full runnable code:

```
const numbers = [1, 2, 3, 4];

const doubled = numbers.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6, 8]
```

You can add parameter types for clarity:

```
const doubled = numbers.map((num: number): number => num * 2);
```

4.4.4 Lexical `this` Binding

One of the most important features of arrow functions is that they **do not create their own `this` context**. Instead, they **capture `this` from the surrounding scope**.

This makes arrow functions very useful in scenarios where traditional functions would lead to confusing or incorrect `this` behavior—like inside classes or event handlers.

Example: Arrow function vs traditional function in a class

```
class Counter {
  count = 0;

  // Using arrow function to preserve `this`
  start() {
    setInterval(() => {
      this.count++;
      console.log(this.count);
    }, 1000);
  }
}
```

If we used a regular function in `setInterval`, `this` would refer to the global object (or be `undefined` in strict mode), not the instance of the class.

4.4.5 Summary of Differences

Feature	Traditional Function	Arrow Function
Syntax	Longer	Shorter, more concise
`this` binding	Own `this` (dynamic)	Lexical `this` (inherits from surrounding scope)
Used in classes/callbacks	Can be tricky due to dynamic `this`	Easier to use in callbacks and methods
Return behavior	Requires `return` and `{}` for blocks	Supports implicit return for single expressions

4.4.6 Final Example: Arrow Function in Practice

Full runnable code:

```
const names = ["Alice", "Bob", "Charlie"];

const greetings = names.map(name => `Hello, ${name}!`);

console.log(greetings);
// Output: [ 'Hello, Alice!', 'Hello, Bob!', 'Hello, Charlie!' ]
```

Arrow functions make this code clean, readable, and easy to understand.

Arrow functions are a powerful tool in TypeScript that simplify function expressions and eliminate common issues with `this` binding. Use them for concise syntax, especially in callbacks, array operations, and when working inside classes.

Chapter 5.

Object Types and Type Aliases

1. Object Shape Declarations
2. Optional Properties and Readonly
3. Using `type` and `interface`

5 Object Types and Type Aliases

5.1 Object Shape Declarations

In TypeScript, you can define the **shape of an object**—that is, what properties it must have and the types of those properties. This allows you to write safer, more predictable code and catch errors when objects don't match the expected structure.

5.1.1 Inline Object Type Annotations

The simplest way to define an object type is with an **inline annotation**. You specify the object's property names and types directly in the variable declaration.

Example: Inline object type

```
let user: { name: string; age: number };

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

This tells TypeScript that `user` must be an object with:

- a `name` property of type `string`
- an `age` property of type `number`

Trying to assign an object that doesn't match this shape will result in a type error:

```
user = {
  name: "Bob"
  // Error: Property 'age' is missing
};
```

5.1.2 Functions Accepting Objects with Specific Properties

You can annotate parameters to accept only objects that match a certain shape.

Example: Function with object parameter

Full runnable code:

```
function printUserInfo(user: { name: string; age: number }) {
  console.log(`${user.name} is ${user.age} years old.`);
}
```

```
printUserInfo({ name: "Charlie", age: 25 }); // YES OK
```

Example: Error from missing or extra properties

```
printUserInfo({ name: "Dana" });
// NO Error: Property 'age' is missing

printUserInfo({ name: "Eve", age: 40, email: "eve@example.com" });
// NO Error: Object literal may only specify known properties
```

TypeScript is strict about **object literals** passed directly into functions. It will raise an error for **excess properties**, unless handled explicitly.

5.1.3 Handling Extra Properties: Type Compatibility

You can avoid excess property errors by assigning the object to a variable first. TypeScript performs **structural type checking**, so it will allow extra properties if the object is stored in a variable.

```
const person = { name: "Eve", age: 40, email: "eve@example.com" };
printUserInfo(person); // YES OK
```

5.1.4 Optional and Flexible Shapes (Preview)

To explicitly allow extra or dynamic properties, you can use techniques like **index signatures**, which you'll learn about in upcoming sections.

Full runnable code:

```
function logConfig(config: { [key: string]: string }) {
  for (const key in config) {
    console.log(`${key}: ${config[key]}`);
  }
}

logConfig({ theme: "dark", layout: "grid" }); // YES OK
```

5.1.5 Summary

Concept	Example	Notes
Inline object type	`{ name: string; age: number }`	Defines the exact shape directly
Function parameter typing	`function(user: { name: string })`	Enforces object structure when calling functions
Excess property check	Direct object literals are strictly checked	Extra fields not in the shape cause errors
Type compatibility	Assigned variables allow extra fields	TypeScript uses structural (duck) typing

Defining object shapes allows you to write functions and variables that **expect structured data**, making your code more robust and self-documenting. In the next sections, you'll learn how to add **optional**, **readonly**, and **flexible** properties to further refine your object types.

5.2 Optional Properties and Readonly

TypeScript gives you tools to model **flexible and safe object structures** through optional properties (`?`) and immutable properties (`readonly`). These features help you write more precise types and catch mistakes at compile time.

5.2.1 Optional Properties (`?`)

An **optional property** is one that may or may not exist on an object. It's defined by adding a `?` after the property name in the type declaration.

Example: User profile with optional `email`

```
type UserProfile = {
  username: string;
  age: number;
  email?: string;  // optional
};

const user1: UserProfile = {
  username: "alice",
  age: 28
};

const user2: UserProfile = {
  username: "bob",
  age: 35,
  email: "bob@example.com"
};
```

- `email` is optional — you can omit it.
- TypeScript will **not** raise an error if the property is missing.
- However, accessing it requires checking if it's defined:

```
function printEmail(user: UserProfile) {
  if (user.email) {
    console.log(`Email: ${user.email}`);
  } else {
    console.log("No email provided.");
  }
}
```

5.2.2 Readonly Properties

Use the `readonly` modifier to make a property **immutable** after it's assigned. This prevents accidental changes and enforces safer coding patterns.

Example: Readonly configuration object

```
type Config = {
  readonly appName: string;
  readonly version: number;
  debugMode: boolean;
};

const config: Config = {
  appName: "MyApp",
  version: 1.0,
  debugMode: true
};

config.debugMode = false;      // YES OK
// config.appName = "NewName"; // NO Error: Cannot assign to 'appName' because it is a read-only proper
```

When to Use `readonly`

Use `readonly` for:

- **Constants** that should never change after initialization
- Preventing accidental modifications to **configurations** or **identifiers**
- **Defensive programming**: protect data structures from being mutated unintentionally

5.2.3 Combining `readonly` and `?`

You can use `readonly` with optional properties as well:

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

```
  name: string;
  readonly email?: string;
};

const u: ImmutableUser = { id: 101, name: "Jane" };
// u.id = 102;          // NO Error: Cannot assign to 'id'
// u.email = "j@x.com" // NO Error: Cannot assign to optional readonly 'email'
```

5.2.4 Summary

Feature	Syntax	Behavior
Optional	`prop?: type`	Property may be undefined or omitted entirely
Read-only	`readonly prop: type`	Property can't be reassigned after initialization
Combined	`readonly prop?: type`	Optional property that, if present, is immutable

By using `?` and `readonly`, you create object types that clearly express **what is required, what is optional, and what must not change**. This results in code that's safer, easier to understand, and better aligned with the real-world structure of your data.

In the next section, you'll learn how to create **named reusable object types** using `type` and `interface`.

5.3 Using `type` and `interface`

In TypeScript, both `type` and `interface` can be used to define custom object types. While they often serve similar purposes, there are important differences in **syntax**, **extension**, and **features**. Understanding when to use each will help you write cleaner, more maintainable code.

5.3.1 Defining Object Types

Both `type` and `interface` can describe the shape of an object.

Using `type`

```
type User = {
  username: string;
```

```
  age: number;
};
```

Using `interface`

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

Both define an object with `username` (string) and `age` (number).

5.3.2 Using in Functions and Classes

You can use either `type` or `interface` to annotate function parameters or class implementations.

Example: Function parameter

```
function greet(user: User): void {
  console.log(`Hello, ${user.username}!`);
}
```

Example: Implementing an interface in a class

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

class ConsoleLogger implements Logger {
  log(message: string) {
    console.log(`[LOG]: ${message}`);
  }
}
```

Interfaces work well for defining class contracts.

5.3.3 Extending and Combining Types

Extending with `interface` (using `extends`)

```
interface Person {
  name: string;
}

interface Employee extends Person {
  employeeId: number;
```

```
}

const emp: Employee = {
  name: "Alice",
  employeeId: 123
};
```

Combining with `type` (using intersections)

```
type Person = {
  name: string;
};

type Employee = Person & {
  employeeId: number;
};

const emp: Employee = {
  name: "Bob",
  employeeId: 456
};
```

- `interface` uses `extends` for inheritance.
- `type` uses `&` to create intersections (merge types).

5.3.4 Key Differences: `type` vs `interface`

Feature	`interface`	`type`
Extending/Inheritance	YES `extends` another interface	YES Combine with `&` (intersection)
Declaration merging	YES Yes (can define the same interface twice)	NO No (duplicate names cause error)
Used for functions, unions, etc.	NO Not directly	YES Yes (e.g., 'type Result = string | number')
Preferred for classes/OOP	YES Yes	YES Possible but less idiomatic
Literal types and tuples	NO Not supported	YES Fully supported

5.3.5 Declaration Merging (Interfaces Only)

Interfaces support **declaration merging**, where multiple declarations with the same name are automatically combined.

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

interface Animal {
  age: number;
}

const dog: Animal = {
  name: "Rex",
  age: 4
};
```

This can be useful for extending interfaces from libraries or across modules.

With `type`, this would cause a **duplicate identifier error**:

```ts
type Animal = { name: string };
type Animal = { age: number }; // NO Error: Duplicate identifier
```

5.3.6 Which One Should You Use?

Use Case	Recommendation
Object/class structure	Prefer `interface`
Union or intersection types	Use `type`
Function or tuple types	Use `type`
Declaration merging needed	Use `interface`
One-off inline type usage	Either works

TypeScript's own style guide suggests **favoring `interface`** for public APIs and extending class-like structures, but both are valid and powerful.

5.3.7 Final Example: Combined Usage

Full runnable code:

```ts
type ID = string | number;

interface Person {
  id: ID;
  name: string;
}

interface Employee extends Person {
```

```
  role: string;
}

function printEmployee(emp: Employee) {
  console.log(`${emp.name} [${emp.role}] - ID: ${emp.id}`);
}

printEmployee({ id: "emp123", name: "Alice", role: "Engineer" });
```

5.3.8 Summary

- Use `interface` when defining **object shapes**, especially for **class-based structures** or **public APIs**.
- Use `type` for **unions**, **tuples**, or more **flexible compositions**.
- Both can describe object types, but only `interface` supports **declaration merging**.
- You can often choose based on style or team preference—but understanding the differences helps in making the best choice.

In the next chapter, you'll dive into **type narrowing**, where TypeScript intelligently figures out the type of a variable based on conditions and logic.

Chapter 6.

Union, Intersection, and Literal Types

1. Working with Union Types
2. Type Narrowing and Type Guards
3. Discriminated Unions
4. Literal and Enum-like Types

6 Union, Intersection, and Literal Types

6.1 Working with Union Types

In TypeScript, **union types** let you declare a variable or parameter that can hold **more than one type**. This is useful when a value could come in different forms but still needs to be handled safely and explicitly.

Union types increase flexibility **without sacrificing type safety**.

6.1.1 What Is a Union Type?

A **union type** is written using the vertical bar (`|`) between types:

```
type StringOrNumber = string | number;
```

This means a variable of type `StringOrNumber` can be **either a `string` or a `number`**, but **not both at the same time**.

6.1.2 Example 1: Function Accepting a String or Number

Here's a function that accepts either a string or a number and handles both cases safely:

Full runnable code:

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

printId("abc123"); // Output: Your ID is: ABC123
printId(456.78);   // Output: Your ID is: 456.78
```

- The function parameter `id` can be either `string` or `number`.
- TypeScript **narrows the type** inside the `if` blocks using `typeof`, so you can safely use string or number methods.

6.1.3 Example 2: Variable That Can Be Boolean or Null

```
let isLoggedIn: boolean | null = null;

isLoggedIn = true;  // YES Valid
isLoggedIn = false; // YES Valid
isLoggedIn = null;  // YES Valid
// isLoggedIn = "yes"; // NO Error: Type '"yes"' is not assignable to type 'boolean | null'
```

This pattern is helpful when representing **optional or unset states**, such as form inputs or authentication flags.

6.1.4 Why Use Union Types?

Union types help when:

- A function or variable can take **multiple valid types**
- You want to model **real-world flexibility** (e.g. optional values, different ID types)
- You want to keep **type safety**, ensuring only the allowed types are used

More Examples:

A function that accepts a number or a string: Full runnable code:

```
function double(input: number | string): string {
  if (typeof input === "number") {
    return (input * 2).toString();
  } else {
    return input + input;
  }
}

console.log(double(5));     // "10"
console.log(double("Hi"));  // "HiHi"
```

6.1.5 Summary

Concept	Example Syntax	Description	
Union type	‘string	number‘	Value can be a string **or** a number
Used in parameters	‘function(x: A	B)‘	Allows multiple input types
Used in variables	‘let val: T	null‘	Allows optional or nullable values

Concept	Example Syntax	Description
Safe handling	Use `typeof`, `if`, or guards	TypeScript narrows the type automatically

Union types allow you to write **flexible and safe code** that handles a variety of input shapes. In the next section, you'll learn how **TypeScript narrows types** automatically using checks—this is called **type narrowing** and it builds on what you learned here.

6.2 Type Narrowing and Type Guards

When you use **union types** in TypeScript, the compiler knows a value could be one of several types—but it doesn't assume which one until you **narrow** it down. **Type narrowing** means using control flow and checks in your code to tell TypeScript exactly what type a value is at a given point.

TypeScript uses **type guards**—expressions that perform these checks—to narrow down the type and allow **safe access to properties and methods**.

6.2.1 Why Type Narrowing Is Important

Consider a function with a union type parameter:

```
function printValue(value: string | number) {
  console.log(value.toUpperCase()); // NO Error: Property 'toUpperCase' does not exist on type 'string
}
```

TypeScript doesn't allow calling `toUpperCase()` until you're sure `value` is a string. This is where narrowing comes in.

6.2.2 Type Guard: `typeof`

The `typeof` operator is a common way to narrow primitive types like `string`, `number`, or `boolean`.

Example: Narrowing with `typeof`

```
function printValue(value: string | number) {
  if (typeof value === "string") {
    console.log("String value:", value.toUpperCase());
  } else {
```

```
    console.log("Number value:", value.toFixed(2));
  }
}
```

TypeScript infers that:

- Inside the `if`, `value` is a `string`
- Inside the `else`, it's a `number`

6.2.3 Type Guard: `instanceof`

Use `instanceof` to narrow **class-based types** like `Date`, `Error`, or custom classes.

Example: Narrowing with `instanceof`

```
function format(input: string | Date): string {
  if (input instanceof Date) {
    return input.toISOString();
  } else {
    return input.trim();
  }
}
```

Here, TypeScript understands that:

- `input.toISOString()` is safe for `Date`
- `input.trim()` is safe for `string`

6.2.4 Type Guard: Equality Checks

You can also narrow using direct comparisons or equality checks.

Example: Narrowing with equality

```
function process(value: string | null) {
  if (value !== null) {
    console.log("Length:", value.length); // Safe: value is string
  } else {
    console.log("No value provided.");
  }
}
```

This is useful for optional values (`string | undefined`, `boolean | null`, etc.).

6.2.5 Example: Multiple Types with Safe Handling

```
function handleInput(input: string | number | boolean) {
  if (typeof input === "string") {
    console.log("Uppercased:", input.toUpperCase());
  } else if (typeof input === "number") {
    console.log("Doubled:", input * 2);
  } else {
    console.log("Boolean value:", input ? "true" : "false");
  }
}
```

- `toUpperCase()` is only called on strings.
- Numeric operations only run on numbers.
- Booleans are handled with a conditional.

6.2.6 Custom Type Guards (Preview)

You can also write **custom type guard functions** to help narrow complex types. Here's a sneak peek:

```
function isString(x: unknown): x is string {
  return typeof x === "string";
}
```

You'll learn more about this in advanced chapters.

6.2.7 Summary

Type Guard	Usage	Best For
`typeof`	`typeof x === "string"`	Primitives: string, number, boolean
`instanceof`	`x instanceof ClassName`	Class instances like Date, Error
Equality Checks	`x !== null`, `x === undefined`	Nullable or optional values

By using **type guards**, you can write safe, expressive code that works with **union types**. TypeScript will infer the correct type for you, making your code safer and smarter.

In the next section, you'll build on this by creating **discriminated unions**, which let you narrow complex types even more reliably using common properties.

6.3 Discriminated Unions

Discriminated unions (also called **tagged unions** or **algebraic data types**) are a powerful feature in TypeScript that combine **union types**, **literal types**, and **type narrowing** to make complex logic safer and more predictable.

They work by giving each variant of a union a **shared, literal property**—usually called `kind`, `type`, or `tag`—which can be checked at runtime to determine the exact shape of the object.

6.3.1 Why Use Discriminated Unions?

Discriminated unions help you model multiple related object types in a **type-safe** way, and write code that handles each case clearly and safely.

6.3.2 Example: Shape Types with a `kind` Property

Let's define several shape types that each have a unique `kind` value:

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

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

type Rectangle = {
  kind: "rectangle";
  width: number;
  height: number;
};

type Shape = Circle | Square | Rectangle;
```

Each object type in the union has a unique `kind` string literal. The union `Shape` can now represent any one of these.

6.3.3 Handling Discriminated Unions with `switch`

You can use a `switch` statement to safely handle each shape based on its `kind`. TypeScript will automatically narrow the type for you.

```
function getArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;

    case "square":
      return shape.side * shape.side;

    case "rectangle":
      return shape.width * shape.height;

    default:
      // Exhaustive check
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}
```

Notes:

- Inside each `case`, TypeScript knows exactly which shape you're working with.
- The `default` case checks that all possibilities were handled using the `never` type.
- If a new shape type is added to the `Shape` union but not handled in the switch, TypeScript will show an error at the `never` line.

6.3.4 Benefits of Discriminated Unions

YES **Safer branching logic**: TypeScript narrows the type based on the `kind` field. YES **Exhaustive checking**: You can catch missing cases at compile time. YES **Self-documenting code**: The union defines all the valid shapes explicitly.

6.3.5 Realistic Usage: UI State Example

Discriminated unions aren't just for geometry—they're also useful in application state, like UI views or API responses.

```
type LoadingState = { status: "loading" };
type SuccessState = { status: "success"; data: string };
type ErrorState = { status: "error"; message: string };

type UIState = LoadingState | SuccessState | ErrorState;

function render(state: UIState): void {
  switch (state.status) {
    case "loading":
      console.log("Loading...");
      break;
    case "success":
```

```
      console.log("Data:", state.data);
      break;
    case "error":
      console.log("Error:", state.message);
      break;
  }
}
```

6.3.6 Summary

Concept	Description
Discriminated union	A union of types with a shared literal property (like `kind`)
Type narrowing	TypeScript auto-detects the exact type based on `kind`
Exhaustive checking	Ensure all cases are handled using `never` in the `default` block

Discriminated unions are one of TypeScript's most powerful tools for modeling complex logic in a safe, readable, and maintainable way. Up next, you'll learn how to use **literal types** and **enum-like patterns** to further constrain and control your data.

6.4 Literal and Enum-like Types

Literal types in TypeScript allow you to restrict a variable to **a specific set of values**, rather than broad types like `string` or `number`. They are especially useful when modeling fixed options, such as sizes, modes, or statuses.

Literal types can act as **type-safe substitutes for enums** in many cases, offering a clean and expressive way to define limited sets of values.

6.4.1 What Are Literal Types?

A **literal type** represents an exact value rather than a general type.

Example: String literal types

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

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

```
setSize('medium'); // YES OK
setSize('extra-large'); // NO Error: Argument of type '"extra-large"' is not assignable to type 'Size'.
```

Here, the `Size` type can only be one of the three string values: `'small'`, `'medium'`, or `'large'`.

6.4.2 Literal Types for Type-Safe APIs

Literal types are ideal for APIs or configurations where **only specific values** are valid.

Example: Button configuration

Full runnable code:

```
type ButtonVariant = 'primary' | 'secondary' | 'danger';

function createButton(variant: ButtonVariant) {
  console.log(`Rendering a ${variant} button`);
}

createButton('primary'); // YES
createButton('info');    // NO Error: 'info' is not a valid ButtonVariant
```

6.4.3 Literal Number Types

Literal types can also be **numbers**, though this is less common.

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

function rollDice(): DiceRoll {
  return Math.floor(Math.random() * 6 + 1) as DiceRoll;
}
```

This ensures only values `1` through `6` are considered valid.

6.4.4 Enum-Like Behavior with Literal Types

In many cases, unioned string literals can serve as a **lightweight alternative to enums**, especially when you don't need advanced features like auto-incrementing or reverse mapping.

Example: Mode settings

```
type Mode = 'light' | 'dark' | 'system';

function setMode(mode: Mode) {
  console.log(`Mode set to: ${mode}`);
}
```

This pattern is simpler than using `enum`, and often preferred in modern TypeScript code because it's **easier to read, tree-shakable, and type-safe**.

6.4.5 Bonus: Using Literal Types in Objects

Literal types work great as values or constraints inside object types:

```
type Alert = {
  type: 'success' | 'error' | 'info';
  message: string;
};

const alert1: Alert = {
  type: 'error',
  message: 'Something went wrong.'
};
```

6.4.6 Benefits of Literal Types

YES Constrain values to a predefined set YES Improve type safety in functions and APIs YES Avoid runtime bugs from invalid strings or numbers YES Lightweight alternative to enums YES Fully compatible with union types and narrowing

6.4.7 Summary

Concept	Example	Pur-pose		
String literal type	‘‘start’	‘stop’	‘pause’‘	Restricts value to known strings
Number literal type	‘1	2	3‘	Restricts to exact numbers
Enum-like substitute	‘type Direction = ‘up’	‘down’‘	Provides a simpler enum alternative	

Literal types are a great way to bring **clarity and safety** to your code. In the next chapter, you'll explore how functions and expressions can work with these types to build expressive, strongly-typed behaviors.

Chapter 7.

Enums and Const Enums

1. Numeric and String Enums
2. Reverse Mapping
3. Const Enums and Performance

7 Enums and Const Enums

7.1 Numeric and String Enums

In TypeScript, **enums** are used to define a set of **named constants**. They make your code more readable, self-documenting, and type-safe—especially when you're dealing with fixed sets of values like roles, states, or modes.

There are two main types of enums:

- **Numeric enums**
- **String enums**

7.1.1 Why Use Enums?

Enums help when you want to:

- Group related constant values under a common name
- Avoid magic strings or numbers in your code
- Write expressive, maintainable code with type safety

7.1.2 Numeric Enums

Numeric enums are the default in TypeScript. When you define a numeric enum, TypeScript assigns **auto-incrementing numbers** starting from `0` (unless you provide a custom value).

Example: User Roles

```
enum UserRole {
  Admin,   // 0
  User,    // 1
  Guest    // 2
}

let role: UserRole = UserRole.Admin;
console.log(role); // 0
```

You can also assign custom starting numbers:

```
enum UserRole {
  Admin = 1,
  User,       // 2
  Guest       // 3
}
```

Using Numeric Enums in Functions

Full runnable code:

```
function getPermissions(role: UserRole): string {
  switch (role) {
    case UserRole.Admin:
      return "Full access";
    case UserRole.User:
      return "Limited access";
    case UserRole.Guest:
      return "Read-only access";
    default:
      return "Unknown role";
  }
}

console.log(getPermissions(UserRole.User)); // Limited access
```

7.1.3 String Enums

String enums associate each member with a **specific string value**. These are more readable and are useful when the values are meaningful strings (e.g., for logging or storing in a database).

Example: UI Themes

Full runnable code:

```
enum Theme {
  Light = "light",
  Dark = "dark"
}

let theme: Theme = Theme.Dark;
console.log(theme); // "dark"
```

Unlike numeric enums, string enums **require explicit values** for each member.

Using String Enums in Functions

Full runnable code:

```
enum Theme {
  Light = "light",
  Dark = "dark"
}

function applyTheme(theme: Theme) {
  console.log(`Applying ${theme} mode...`);
}
```

```
applyTheme(Theme.Light); // Applying light mode...
```

7.1.4 Comparing Numeric vs String Enums

Feature	Numeric Enums	String Enums
Default behavior	Auto-incremented numbers (0, 1, 2…)	Requires explicit string values
Readability	Less human-readable at runtime	More readable and descriptive
Reverse mapping	YES Supported	NO Not supported
Common use cases	Internal roles, statuses	Configuration, labels, API identifiers

7.1.5 Summary

- Use **numeric enums** when you need lightweight identifiers or internal values.
- Use **string enums** when the values need to be **readable**, stored, or serialized.
- Enums make code **cleaner, safer, and more maintainable**, especially when combined with control structures like `switch`.

```
enum Status {
  Success = "success",
  Failure = "failure"
}

function handleStatus(status: Status) {
  if (status === Status.Success) {
    console.log("Operation succeeded.");
  } else {
    console.log("Operation failed.");
  }
}
```

In the next section, you'll learn about **reverse mapping**, a unique feature of numeric enums that allows you to look up names by values.

7.2 Reverse Mapping

TypeScript's **numeric enums** have a special feature called **reverse mapping**. This means you can use the enum's numeric value to get back the corresponding enum **key name**.

7.2.1 What Is Reverse Mapping?

When you define a numeric enum, TypeScript generates a **two-way map**:

- From key to numeric value
- From numeric value back to key name

This allows you to retrieve the **name of the enum member** from its numeric value at runtime.

7.2.2 Example of Reverse Mapping

Full runnable code:

```
enum UserRole {
  Admin = 1,
  User,
  Guest
}

console.log(UserRole.Admin);    // Output: 1   (key → value)
console.log(UserRole[1]);       // Output: "Admin" (value → key)
console.log(UserRole[2]);       // Output: "User"
console.log(UserRole[3]);       // Output: "Guest"
```

Here:

- `UserRole.Admin` gives you the numeric value `1`.
- `UserRole[1]` gives you the string `"Admin"` — the key name.

7.2.3 Why Reverse Mapping Isnt Available with String Enums

String enums **do not have reverse mapping** because their values are strings, and **different keys can share the same string value** or have values that aren't valid as keys.

Example:

Full runnable code:

```
enum Theme {
  Light = "light",
  Dark = "dark"
}

console.log(Theme.Light);    // Output: "light"
// console.log(Theme["light"]); // Error: Property 'light' does not exist on type 'typeof Theme'.
```

Since the enum values are strings, there is no safe, automatic way for TypeScript to map back from `"light"` to `"Light"`.

7.2.4 Summary

Feature	Numeric Enums	String Enums
Forward mapping	Key → Number	Key → String
Reverse mapping	Number → Key (available)	Not available
Use case	When two-way lookup is useful	When meaningful string values are needed

Reverse mapping can be helpful for debugging, serialization, or when you need to display the name corresponding to an enum value.

7.3 Const Enums and Performance

TypeScript provides a special kind of enum called a **`const enum`** designed to improve performance by **inlining enum values** directly into the generated JavaScript code. This means **no extra code** is emitted for the enum itself, resulting in smaller and faster output.

7.3.1 What Is a `const enum`?

A `const enum` tells the TypeScript compiler to **replace all references** to its members with the actual values at compile time, instead of generating an object to represent the enum.

7.3.2 How Does It Differ From Regular Enums?

Regular Enum

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

```
}

const move = Direction.Up;
console.log(move); // Output: 0
```

Generated JavaScript (simplified):

```
var Direction;
(function (Direction) {
  Direction[Direction["Up"] = 0] = "Up";
  Direction[Direction["Down"] = 1] = "Down";
  Direction[Direction["Left"] = 2] = "Left";
  Direction[Direction["Right"] = 3] = "Right";
})(Direction || (Direction = {}));

const move = Direction.Up;
console.log(move);
```

The enum generates a full object with both forward and reverse mappings.

`const enum`

Full runnable code:

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

const move = Direction.Up;
console.log(move); // Output: 0
```

Generated JavaScript:

```
const move = 0;
console.log(move);
```

Here, `Direction.Up` is **inlined as the number `0`**, and no enum object exists in the emitted code.

7.3.3 Benefits of `const enum`

- **Smaller output:** No enum object is generated.
- **Faster runtime:** No lookup needed; values are directly embedded.
- **Ideal for performance-sensitive code:** Especially useful in browser or embedded environments.

7.3.4 Limitations of `const enum`

- **No reverse mapping:** Since no enum object exists at runtime, you cannot map values back to keys.
- **Module compatibility:** In some module systems or when using Babel to transpile, `const enum` inlining may not work properly without extra configuration.
- **Debugging:** Since values are inlined, debugging can be less straightforward because the enum name doesn't appear in the compiled output.

7.3.5 When to Use `const enum`

- When you want **maximum runtime performance** and minimal code size.
- When you **do not need reverse mapping** or runtime access to the enum as an object.
- When targeting environments like browsers where bundle size matters.
- When your build setup supports TypeScript's `const enum` inlining (e.g., `tsc` or compatible bundlers).

7.3.6 Summary

Feature	Regular Enum	Const Enum
Code generation	Generates enum object	Inlines values at compile time
Reverse mapping	Available	Not available
Debugging	Easier (object exists)	Harder (values inlined)
Performance	Slower due to lookup	Faster with direct inlining
Best use cases	When you need runtime enum object	When optimizing bundle size/performance

Using `const enum` is a simple and effective way to optimize your TypeScript applications. Just remember its limitations and use it where appropriate!

Chapter 8.

Type Assertion and Type Casting

1. When and How to Use Type Assertions
2. TypeScript and DOM Interactions
3. The `as` Keyword

8 Type Assertion and Type Casting

8.1 When and How to Use Type Assertions

In TypeScript, **type assertion** is a way to tell the compiler:

> *"Trust me, I know better — treat this value as this specific type."*

Unlike type casting in some other languages, **type assertions do not change the runtime value** or perform any special checks. They only affect TypeScript's static type system during compilation.

8.1.1 Why Use Type Assertions?

Sometimes, TypeScript's type inference or analysis isn't enough to determine the exact type, especially when:

- You receive values of type `unknown` or `any` (for example, from external APIs).
- Working with DOM elements where TypeScript only knows a general type.
- You want to narrow the type of a variable to a more specific one based on your knowledge.

8.1.2 Basic Syntax

There are two common ways to assert a type:

```
// Angle-bracket syntax
let someValue: unknown = "Hello TypeScript";
let strLength: number = (<string>someValue).length;

// 'as' syntax (preferred in JSX and modern code)
let someValue2: unknown = "Hello again";
let strLength2: number = (someValue2 as string).length;
```

Both tell TypeScript to treat `someValue` as a `string`, allowing access to string properties like `.length`.

8.1.3 Example: Asserting an `unknown` or `any` Value

Full runnable code:

```ts
function handleData(data: unknown) {
  // We know 'data' should be a string here
  const message = (data as string).toUpperCase();
  console.log(message);
}

handleData("hello"); // Outputs: HELLO
```

Without the assertion, TypeScript would not allow calling `.toUpperCase()` on `unknown`.

8.1.4 Example: Narrowing DOM Elements

When working with the DOM, TypeScript often types elements as general `HTMLElement` or `Element`, but you may know it's a specific element type like `HTMLInputElement`.

```ts
const input = document.querySelector("#username");

// Assert to HTMLInputElement to access .value
const userInput = (input as HTMLInputElement).value;
console.log(userInput);
```

Without this assertion, TypeScript would complain because `value` is not a property of the general `Element` type.

8.1.5 Caution: Use Assertions Responsibly

- **Assertions bypass TypeScript's type checks.** If you assert incorrectly, you may introduce runtime errors.
- Avoid using assertions as a shortcut to silence errors without verifying the type.
- When possible, prefer **type guards** or proper validation over assertions.

8.1.6 Summary

What Type Assertion Does	What It Does NOT Do
Tell TypeScript to treat a value as a specific type	Change or convert the value at runtime
Enable accessing properties or methods	Perform runtime type checking
Help when compiler cannot infer the correct type	Guarantee safety if used incorrectly

Use type assertions when you are **certain about a value's type** but TypeScript cannot

infer it automatically. This helps you write clearer code while keeping the runtime safe and predictable.

8.2 TypeScript and DOM Interactions

When working with the **DOM** in TypeScript, you often interact with elements through methods like `getElementById` or `querySelector`. However, TypeScript usually infers these elements as general types like `HTMLElement` or even `Element`, which don't always include specific properties you need, such as `.value` on an `<input>` element.

This is where **type assertions** become essential to tell TypeScript exactly what kind of element you're dealing with.

8.2.1 Why Are Assertions Needed?

For example, `document.getElementById` returns an `HTMLElement | null` because the element might not exist or could be any kind of element. But if you know the element is an `<input>`, you need to assert its type to safely access `.value`.

8.2.2 Using Assertions with `getElementById`

Full runnable code:

```
// Without assertion - TypeScript only knows this is an HTMLElement or null
const inputElement = document.getElementById("username");

// Using assertion to tell TypeScript it's an HTMLInputElement
const input = inputElement as HTMLInputElement | null;

if (input) {
  // Now it's safe to access .value because input is HTMLInputElement
  console.log(input.value);
} else {
  console.log("Element not found");
}
```

Here, asserting with `as HTMLInputElement` allows accessing `.value` without a compiler error.

8.2.3 Using Assertions with `querySelector`

`querySelector` returns `Element | null` by default, which doesn't have `.value` or `.textContent`.

Full runnable code:

```
// Select a div element
const divElement = document.querySelector("#content") as HTMLDivElement | null;

if (divElement) {
  // Safe access to .textContent property
  console.log(divElement.textContent);
}
```

You can assert to other specific element types like `HTMLButtonElement`, `HTMLAnchorElement`, etc., depending on the element you expect.

8.2.4 Full Example: Form Input and Button

```
const input = document.getElementById("email") as HTMLInputElement | null;
const submitButton = document.querySelector("button.submit") as HTMLButtonElement | null;

if (input && submitButton) {
  submitButton.addEventListener("click", () => {
    alert(`Email entered: ${input.value}`);
  });
}
```

This pattern is common for safe interaction with specific DOM elements.

8.2.5 Summary

- DOM methods often return generic element types or `null`.
- Use **type assertions** to specify the exact element type (e.g., `HTMLInputElement`).
- Assertions allow safe access to element-specific properties like `.value` and `.textContent`.
- Always check for `null` before accessing properties to avoid runtime errors.

Type assertions help bridge the gap between TypeScript's static type system and the dynamic nature of the DOM, making your DOM manipulations safer and error-free.

8.3 The as Keyword

In TypeScript, the **as keyword** is the most common and recommended way to perform **type assertions**—telling the compiler to treat a value as a specific type.

8.3.1 Syntax of the as Keyword

The syntax looks like this:

```
let someValue: unknown = "Hello, TypeScript!";
let strLength: number = (someValue as string).length;
```

Here, `(someValue as string)` asserts that `someValue` is of type `string`, allowing access to string-specific properties like `.length`.

8.3.2 Comparison with Angle-Bracket Syntax

Before the `as` syntax became common, TypeScript supported **angle-bracket assertions**:

```
let someValue: unknown = "Hello, TypeScript!";
let strLength: number = (<string>someValue).length;
```

Both forms perform the **same assertion** and have no runtime effect.

8.3.3 Why Prefer as Over Angle Brackets?

1. **JSX Compatibility** In projects using JSX (e.g., React), angle brackets conflict with JSX syntax, causing parsing errors:

   ```
   // This causes an error in JSX files:
   let strLength = (<string>someValue).length;
   ```

   The `as` syntax avoids this problem:

   ```
   let strLength = (someValue as string).length;
   ```

2. **Readability** The `as` syntax is often clearer and visually separates the value from the type.

3. **Consistency** Most modern TypeScript codebases and official documentation use the `as` keyword.

8.3.4 Equivalent Examples

Angle-Bracket Syntax	`as` Keyword Syntax
`let num = (<number>value);`	`let num = value as number;`
`let input = <HTMLInputElement>el;`	`let input = el as HTMLInputElement;`

8.3.5 Summary

- The `as` keyword is the **recommended way** to perform type assertions in TypeScript.
- It is **especially necessary** in JSX/TSX files where angle brackets cause syntax conflicts.
- Both `as` and angle brackets do **not affect runtime behavior**, only compile-time types.
- Using `as` improves code clarity and consistency across TypeScript projects.

Chapter 9.

Control Flow and Type Safety

1. Conditional Statements
2. Loops and Iteration Helpers
3. Exhaustive Checks with `never`

9 Control Flow and Type Safety

9.1 Conditional Statements

Conditional statements like `if`, `else if`, `else`, and `switch` are fundamental tools for controlling program flow. In TypeScript, combining these statements with **type-safe variables** helps you write robust and error-free branching logic.

9.1.1 Using `if`, `else if`, and `else` with Typed Variables

TypeScript's static type system ensures that variables have the expected types, helping catch mistakes early.

Example: Handling String Status

Full runnable code:

```
let status: string = "loading";

if (status === "loading") {
  console.log("Loading data...");
} else if (status === "success") {
  console.log("Data loaded successfully!");
} else if (status === "error") {
  console.log("Error loading data.");
} else {
  console.log("Unknown status.");
}
```

TypeScript helps by:

- Checking you're comparing with strings (`"loading"`, `"success"`, etc.).
- Alerting if you mistype a string or compare incompatible types.

9.1.2 Using `if` with Number Types

```
let score: number = 85;

if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else {
  console.log("Grade: C or below");
}
```

TypeScript ensures you don't accidentally compare numbers to strings or other types.

9.1.3 Using `switch` Statements with Enums

Enums are perfect for switch statements because they define a fixed set of possible values.

Example: Using an Enum for User Roles

Full runnable code:

```
enum UserRole {
  Admin,
  User,
  Guest
}

function getPermissions(role: UserRole) {
  switch (role) {
    case UserRole.Admin:
      return "Full access";
    case UserRole.User:
      return "Limited access";
    case UserRole.Guest:
      return "Read-only access";
    default:
      // TypeScript ensures this case is never reached if all enum values are handled
      return "Unknown role";
  }
}

console.log(getPermissions(UserRole.User)); // Limited access
```

9.1.4 How TypeScript Improves Conditional Logic Safety

- **Type checks in comparisons:** You cannot accidentally compare different types.
- **Exhaustiveness checks:** When using enums with `switch`, TypeScript can warn you if you forget to handle a case (especially with the `never` type in more advanced patterns).
- **Code clarity:** Explicit types clarify the expected inputs in conditions, reducing bugs.

9.1.5 Summary

Statement	Use Case
`if` / `else if` / `else`	Flexible, checks conditions with expressions
`switch`	Cleaner when checking a variable against many fixed values (like enums)

By combining these control flow structures with TypeScript's strong typing, you write clearer, safer branching logic and reduce runtime errors.

9.2 Loops and Iteration Helpers

In TypeScript, loops and iteration helpers let you process collections like arrays and objects efficiently — and with **type safety** guaranteed.

9.2.1 Common Loops with Typed Arrays

`for` Loop

The classic `for` loop is useful for iterating with an index.

Full runnable code:

```
const numbers: number[] = [10, 20, 30];

for (let i = 0; i < numbers.length; i++) {
  const num = numbers[i];
  console.log(num * 2); // Safe: num is a number
}
```

TypeScript knows `numbers[i]` is a `number`, so it allows math operations without errors.

`while` Loop

You can use `while` loops safely with typed counters or conditions:

Full runnable code:

```
let count: number = 3;

while (count > 0) {
  console.log(count);
  count--;
}
```

`for...of` Loop

The `for...of` loop is ideal for iterating directly over **values** of iterable collections like arrays:

Full runnable code:

```
const fruits: string[] = ["apple", "banana", "cherry"];

for (const fruit of fruits) {
  console.log(fruit.toUpperCase()); // Safe: fruit is a string
}
```

`for...in` Loop

The `for...in` loop iterates over **keys** (property names) of an object:

Full runnable code:

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

for (const key in person) {
  const value = person[key as keyof typeof person]; // Type assertion to avoid errors
  console.log(`${key}: ${value}`);
}
```

Note: When using `for...in` with objects, TypeScript requires type assertions to ensure keys are valid.

9.2.2 Iteration Helpers on Typed Arrays

Higher-order functions like `.map()`, `.filter()`, and `.forEach()` provide powerful and expressive ways to iterate arrays.

`.map()` Transforming Arrays

Full runnable code:

```
const numbers: number[] = [1, 2, 3];
const doubled = numbers.map(num => num * 2);

console.log(doubled); // Output: [2, 4, 6]
```

- The callback parameter `num` is **inferred** as `number`.
- Return values must match the mapped array's type (`number` here).

`.filter()` Selecting Items

```
const mixedNumbers: number[] = [1, 5, 10, 15];
const bigNumbers = mixedNumbers.filter(num => num > 5);

console.log(bigNumbers); // Output: [10, 15]
```

- The filter function receives `number` and returns a boolean.
- TypeScript guarantees type consistency.

`.forEach()` Side Effects

Full runnable code:

```
const names: string[] = ["Alice", "Bob", "Charlie"];

names.forEach(name => {
  console.log(name.toLowerCase());
});
```

- `name` is typed as `string`.
- No return value is expected.

9.2.3 Summary

Loop Type	Use Case	TypeScript Benefit
`for`	Index-based iteration	Enforces correct indexing and type checking
`while`	Conditional looping	Keeps condition and variables typed
`for...of`	Iterate values of iterable collections	Directly accesses typed values
`for...in`	Iterate keys in objects	Requires type-safe key access
`.map()`, `.filter()`, `.forEach()`	Functional array processing	Callback parameters and return types inferred

By using these looping constructs and helpers, TypeScript helps you write clean, concise, and safe code when working with collections.

9.3 Exhaustive Checks with `never`

In TypeScript, the **`never` type** represents values that **never occur**. It's a powerful tool for ensuring **exhaustive checks** in control flow, particularly when working with union types like discriminated unions.

9.3.1 What Is the `never` Type?

- It indicates a function or expression that **never returns normally** (e.g., throws an error or runs infinitely).
- It's also used as a **type for unreachable code**—places in your code that should never execute.

9.3.2 Using `never` for Exhaustive Checks

When you handle all possible cases of a union type (for example, with a `switch` statement), you can use the `never` type in a `default` case to catch any unhandled cases.

This technique helps:

- Ensure **all variants are handled explicitly**.
- Detect **future bugs or missing cases** when the union is extended.
- Make your code **safer and easier to maintain**.

9.3.3 Example: Exhaustive `switch` with a Discriminated Union

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; sideLength: number }
  | { kind: "rectangle"; width: number; height: number };

function area(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
    case "rectangle":
      return shape.width * shape.height;
    default:
      // The 'exhaustiveCheck' variable has type 'never' here,
      // meaning all cases should have been handled above.
      const exhaustiveCheck: never = shape;
```

```
      throw new Error(`Unhandled shape: ${exhaustiveCheck}`);
  }
}
```

How It Works

- The `default` block assigns `shape` to a variable typed as `never`.
- If **all cases are handled**, `shape` cannot have any other type, so the assignment is valid.
- If a new variant is added to `Shape` but not handled in the `switch`, TypeScript will produce a **compile-time error** here because `shape` will not be `never`.
- This signals to you that you need to update the `switch` statement accordingly.

9.3.4 Benefits of Using `never` for Exhaustiveness

- **Catch missing cases at compile time**, preventing runtime bugs.
- Encourage **complete handling** of all union variants.
- Improve **code clarity and maintainability**.

9.3.5 Summary

Concept	Description
`never` type	Represents impossible or unreachable values
Exhaustive checks	Use `never` in `default` to verify all cases
Control flow safety	Prevents silent bugs when union types evolve

Using `never` for exhaustive checks is a best practice in TypeScript that helps make your control flow logic bulletproof — ensuring your code handles every possible case and remains correct as it grows.

Chapter 10.

Interfaces and Classes

1. Interface Basics and Extension
2. Class Definitions and Inheritance
3. Public, Private, Protected, and `readonly` Modifiers
4. Getters and Setters
5. Implementing Interfaces

10 Interfaces and Classes

10.1 Interface Basics and Extension

Interfaces in TypeScript define **object shapes** and **contracts** that describe the structure and types of data. They help ensure that objects adhere to a particular design, making your code more predictable, reusable, and easier to maintain.

10.1.1 What Is an Interface?

An **interface** declares the expected properties and their types for an object, without providing implementation details.

Example: Defining a Simple `Person` Interface

Full runnable code:

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

function greet(person: Person) {
  console.log(`Hello, ${person.name}! You are ${person.age} years old.`);
}

const user: Person = { name: "Alice", age: 30 };
greet(user);
```

- The `Person` interface specifies that a person must have a `name` (string) and an `age` (number).
- The `greet` function requires an argument that fits this `Person` shape.
- Passing any object that matches the interface is allowed, promoting **type safety** and **code clarity**.

10.1.2 Extending Interfaces with `extends`

Interfaces can build upon other interfaces by using `extends`, creating a new interface that combines properties from multiple sources.

Example: Extending `Person` to `Employee`

Full runnable code:

```ts
interface Employee extends Person {
  employeeId: number;
  department: string;
}

const employee: Employee = {
  name: "Bob",
  age: 40,
  employeeId: 12345,
  department: "Sales",
};

function printEmployeeInfo(emp: Employee) {
  console.log(
    `${emp.name} works in ${emp.department} with ID ${emp.employeeId}.`
  );
}

printEmployeeInfo(employee);
```

- The `Employee` interface extends `Person`, inheriting `name` and `age`.
- It adds `employeeId` and `department` properties.
- This allows reuse of the `Person` structure while adding more specific details.

10.1.3 Why Use Interfaces?

- **Code Reusability:** Define common shapes once and extend or implement them elsewhere.
- **Consistency:** Enforce uniform data structures across your application.
- **Documentation:** Serve as clear contracts that communicate expected object properties.
- **Tooling Support:** Enable better code completion and error detection in editors.

10.1.4 Summary

Concept	Description
Interface	Defines the structure (shape) of an object
Function annotation	Use interfaces to type function parameters
Interface extension	Use `extends` to build on existing interfaces

Interfaces are fundamental to building scalable, maintainable TypeScript applications by clearly defining expected object shapes and promoting consistent code architecture.

10.2 Class Definitions and Inheritance

Classes in TypeScript provide a blueprint for creating objects with properties and methods. They support modern object-oriented features like **constructors**, **inheritance**, and **method overriding**.

10.2.1 Defining a Class

A class defines properties and methods. The constructor initializes new instances.

Full runnable code:

```
class Animal {
  name: string;

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

  speak(): void {
    console.log(`${this.name} makes a sound.`);
  }
}

// Creating an instance of Animal
const animal = new Animal("Generic Animal");
animal.speak(); // Output: Generic Animal makes a sound.
```

- The `Animal` class has a `name` property.
- The `constructor` sets the initial `name`.
- The `speak` method outputs a generic message.

10.2.2 Inheritance with `extends`

A subclass can inherit properties and methods from a base class using the `extends` keyword. Subclasses can add new features or override existing methods.

Example: `Dog` Extends `Animal`

Full runnable code:

```
class Dog extends Animal {
  constructor(name: string) {
    super(name); // Call base class constructor
  }
```

```
  // Override speak method
  speak(): void {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog("Buddy");
dog.speak(); // Output: Buddy barks.
```

- `Dog` inherits from `Animal`.
- The `super(name)` call invokes the base class constructor to initialize the `name`.
- The `speak` method is **overridden** to provide dog-specific behavior.

10.2.3 Summary of Key Concepts

Feature	Description
Class	Defines properties and methods
Constructor	Special method to initialize new objects
`extends`	Enables inheritance from a base class
`super()`	Calls the base class constructor or methods
Method overriding	Subclass can replace base class methods with new logic

Classes and inheritance let you model real-world hierarchies with reusable and extendable code. This improves organization and enables polymorphic behavior in your applications.

10.3 Public, Private, Protected, and `readonly` Modifiers

Access modifiers in TypeScript control the **visibility** and **accessibility** of class members (properties and methods). They are essential for **encapsulation**, allowing you to hide internal details and expose only what's necessary.

10.3.1 Access Modifiers Overview

Modifier	Accessible From	Description
`public`	Anywhere (default if none given)	Fully accessible
`private`	Only within the class itself	Hidden outside the class

Modifier	Accessible From	Description
`protected`	Within the class and subclasses	Accessible to derived classes, but not outside

10.3.2 `public` (Default)

By default, all properties and methods are `public` unless specified otherwise.

Full runnable code:

```
class Person {
  public name: string; // explicitly public, but optional
  age: number; // also public by default

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

const p = new Person("Alice", 30);
console.log(p.name); // Accessible
console.log(p.age);  // Accessible
```

10.3.3 `private`

`private` members cannot be accessed or modified from outside the class.

Full runnable code:

```
class Person {
  private ssn: string;

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

  revealSSN() {
    console.log(`SSN: ${this.ssn}`);
  }
}

const p = new Person("123-45-6789");
p.revealSSN();        // Works fine
// console.log(p.ssn); // Error: Property 'ssn' is private and only accessible within class 'Person'.
```

10.3.4 `protected`

`protected` members are like `private`, but accessible inside subclasses.

Full runnable code:

```
class BankAccount {
  protected balance: number;

  constructor(initialBalance: number) {
    this.balance = initialBalance;
  }
}

class SavingsAccount extends BankAccount {
  deposit(amount: number) {
    this.balance += amount;  // Allowed because 'balance' is protected
  }

  getBalance() {
    return this.balance;
  }
}

const account = new SavingsAccount(1000);
account.deposit(500);
console.log(account.getBalance()); // Output: 1500
// console.log(account.balance);    // Error: Property 'balance' is protected and only accessible withi
```

10.3.5 `readonly` Immutable Properties

The `readonly` modifier marks a property as **immutable after initialization**. It can only be assigned during declaration or in the constructor.

Full runnable code:

```
class BankAccount {
  readonly accountId: string;
  protected balance: number;

  constructor(accountId: string, initialBalance: number) {
    this.accountId = accountId;  // Allowed here
    this.balance = initialBalance;
  }

  getAccountId() {
    return this.accountId;
  }
}

const acc = new BankAccount("ABC123", 1000);
console.log(acc.getAccountId()); // Output: ABC123
```

```
// acc.accountId = "XYZ456";      // Error: Cannot assign to 'accountId' because it is a read-only prop
```

10.3.6 Summary

Modifier	Purpose	Example Use Case
`public`	Accessible anywhere (default)	Data or methods meant for public API
`private`	Hide implementation details from outside	Sensitive data like passwords
`protected`	Allow subclass access, hide from outside	Internal state shared with subclasses
`readonly`	Prevent reassignment after initialization	IDs, constants, configuration values

By using access modifiers and `readonly`, you can protect your class internals, enforce immutability where needed, and build safer, more maintainable TypeScript applications.

10.4 Getters and Setters

TypeScript supports **getters** and **setters**, special methods that provide controlled access to class properties. They allow you to **encapsulate** logic for reading or modifying properties while keeping the external interface simple.

10.4.1 What Are Getters and Setters?

- A **getter** (`get`) allows you to define a method that runs when a property is accessed.
- A **setter** (`set`) allows you to define a method that runs when a property is assigned a value.
- Together, they enable validation, transformation, or computed properties while keeping syntax clean.

10.4.2 Practical Example: `Temperature` Class

Let's create a `Temperature` class that stores a temperature internally in Celsius but allows getting and setting it in Fahrenheit.

Full runnable code:

```
class Temperature {
  private _celsius: number;

  constructor(celsius: number) {
    this._celsius = celsius;
  }

  // Getter for Celsius
  get celsius(): number {
    return this._celsius;
  }

  // Setter for Celsius with validation
  set celsius(value: number) {
    if (value < -273.15) {
      throw new Error("Temperature cannot be below absolute zero!");
    }
    this._celsius = value;
  }

  // Getter for Fahrenheit (computed property)
  get fahrenheit(): number {
    return this._celsius * 9 / 5 + 32;
  }

  // Setter for Fahrenheit that updates Celsius internally
  set fahrenheit(value: number) {
    this.celsius = (value - 32) * 5 / 9;  // Reuse celsius setter for validation
  }
}

const temp = new Temperature(25);
console.log(temp.celsius);    // Output: 25
console.log(temp.fahrenheit); // Output: 77

temp.fahrenheit = 86;
console.log(temp.celsius);    // Output: 30

// temp.celsius = -300;       // Throws error: Temperature cannot be below absolute zero!
```

10.4.3 Why Use Getters and Setters?

- **Encapsulation:** Hide internal representation of data (e.g., `_celsius`).
- **Validation:** Validate new values before assigning them.
- **Computed values:** Provide dynamic values derived from stored data.
- **Cleaner syntax:** Access properties like fields but control their behavior internally.

10.4.4 Summary

Feature	Description
`get`	Define a method accessed like a property
`set`	Define a method to validate or transform on assignment
Encapsulation	Protect internal state with private backing fields
Data Integrity	Validate values before updating properties

Getters and setters provide a powerful way to manage class state safely while keeping an intuitive property-based interface for users of your class.

10.5 Implementing Interfaces

In TypeScript, **interfaces** define contracts that classes can **implement** to guarantee they provide certain properties or methods. This enforces a consistent structure across different classes, improving code reliability and interoperability.

10.5.1 How Classes Implement Interfaces

When a class **implements** an interface, it must provide implementations for all the interface's members. TypeScript will give a compile-time error if any required members are missing.

10.5.2 Example: `Drawable` Interface

Let's define a simple interface `Drawable` that requires a `draw()` method:

```
interface Drawable {
  draw(): void;
}
```

10.5.3 Classes Implementing `Drawable`

Here are two different classes that implement the `Drawable` interface:

```
class Circle implements Drawable {
  radius: number;
```

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

  draw(): void {
    console.log(`Drawing a circle with radius ${this.radius}`);
  }
}

class Square implements Drawable {
  sideLength: number;

  constructor(sideLength: number) {
    this.sideLength = sideLength;
  }

  draw(): void {
    console.log(`Drawing a square with side length ${this.sideLength}`);
  }
}
```

Both classes implement the required `draw()` method, fulfilling the `Drawable` contract.

10.5.4 Using Implementations Interchangeably

Because both classes implement the same interface, they can be treated uniformly:

```
function render(shape: Drawable) {
  shape.draw();
}

const circle = new Circle(5);
const square = new Square(10);

render(circle);  // Output: Drawing a circle with radius 5
render(square);  // Output: Drawing a square with side length 10
```

10.5.5 TypeScripts Enforcement

- If a class fails to implement a method from the interface, TypeScript throws a compile error.
- This guarantees that objects of implementing classes are compatible with the interface type.
- This helps with **polymorphism**, where functions can work with any object that implements the interface, regardless of the underlying class.

10.5.6 Summary

Concept	Description
Interface contract	Defines required properties/methods
`implements`	Class must provide all interface members
Polymorphism	Different classes can be used interchangeably via interface types
Compile-time safety	TypeScript enforces correctness at compile time

Implementing interfaces is a powerful way to design flexible, maintainable TypeScript applications with strong type safety and consistent object shapes.

Chapter 11.

Generics

1. Generic Functions and Types
2. Constraints and Defaults
3. Using Generics with Interfaces and Classes
4. Real-World Generic Examples

11 Generics

11.1 Generic Functions and Types

Generics in TypeScript allow you to write **reusable** and **type-safe** code that works with a variety of types, without sacrificing type information. Instead of hardcoding a specific type, you use **type parameters** as placeholders that get filled in when the function or type is used.

11.1.1 Why Use Generics?

- Write functions, classes, or interfaces that work with many types.
- Preserve type safety and avoid the need for `any`.
- Enable better code reuse and flexibility.

11.1.2 Example: Generic Identity Function

Here's a simple function that returns whatever it receives — the identity function.

Without generics, you might lose type information or need to write multiple versions.

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

- `<T>` is a **generic type parameter**.
- `arg: T` means the function accepts a parameter of type `T`.
- The function returns a value of the same type `T`.

11.1.3 Using the Generic Function

TypeScript can often **infer** the type argument based on the call:

Full runnable code:

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

const num = identity(42);       // Type inferred as number
const str = identity("hello");  // Type inferred as string
```

```
console.log(num); // Output: 42
console.log(str); // Output: hello
```

You can also specify the type explicitly if needed:

```
const explicit = identity<number>(100);
```

11.1.4 Generic Functions with Arrays

Generics are particularly useful with collections like arrays. Here's a generic function that returns the first element of an array:

Full runnable code:

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

const nums = [10, 20, 30];
const firstNum = firstElement(nums);  // inferred as number

const words = ["apple", "banana"];
const firstWord = firstElement(words); // inferred as string

console.log(firstNum);  // Output: 10
console.log(firstWord); // Output: apple
```

11.1.5 Summary

Concept	Description
Generic `<T>`	Type parameter placeholder
Generic functions	Write functions reusable with any type
Type inference	TypeScript often infers generic types
Type safety	Keeps type information precise and safe

Generics enable you to create flexible and robust APIs by abstracting over types while retaining full type safety. They are essential for building reusable components in TypeScript.

11.2 Constraints and Defaults

When working with generics, sometimes you want to **restrict** the kinds of types that can be used, or provide a **default** type if none is specified. TypeScript allows this using **constraints** with `extends` and **default generic parameters**.

11.2.1 Generic Constraints with `extends`

A **constraint** limits a generic type to only accept types that satisfy certain conditions. This helps ensure the generic type supports specific properties or methods your function or class relies on.

Example: Constraint on Objects with `.length`

Suppose you want a function that accepts any value that has a `.length` property (like arrays, strings, or objects with length).

Full runnable code:

```
interface HasLength {
  length: number;
}

function logLength<T extends HasLength>(arg: T): T {
  console.log(`Length is: ${arg.length}`);
  return arg;
}

logLength("hello");          // Length is: 5
logLength([1, 2, 3, 4]);     // Length is: 4
logLength({ length: 10 });   // Length is: 10

// logLength(123);           // Error: number doesn't have 'length'
```

Here:

- `<T extends HasLength>` means `T` must be a type that has a `length` property.
- Trying to pass a type without `length` causes a compile-time error.
- This makes the function **safer** because it can safely use `arg.length`.

11.2.2 Default Generic Types

You can also provide a **default type** for generics, so if a caller doesn't specify a type, the default is used.

Example: Generic Class with Default Type

Full runnable code:

```
class Box<T = string> {
  contents: T;

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

  getContents(): T {
    return this.contents;
  }
}

const stringBox = new Box("hello");   // T inferred as string (default)
const numberBox = new Box<number>(123); // T explicitly set to number

console.log(stringBox.getContents()); // Output: hello
console.log(numberBox.getContents()); // Output: 123
```

Here:

- `T = string` sets a default type parameter.
- When the generic type is not specified, `string` is used.
- You can override the default by specifying another type explicitly.

11.2.3 Why Use Constraints and Defaults?

Feature	Benefit
Constraints	Improve type safety by enforcing shape or behavior
Defaults	Simplify usage by providing sensible defaults
Together	Make generic code more robust and user-friendly

11.2.4 Summary

- Use `extends` to **constrain** generic types to those compatible with a specific interface or type.
- Use default generic parameters (`T = SomeType`) to provide fallback types.
- Constraints prevent runtime errors by ensuring the type supports needed properties.
- Defaults reduce verbosity for common cases.

By applying constraints and defaults, you write generic code that's safer, clearer, and easier to use across different scenarios.

11.3 Using Generics with Interfaces and Classes

Generics are not just for functions — they can also be applied to **interfaces** and **classes** in TypeScript. This allows you to build flexible, reusable components that work with a variety of data types while still maintaining **strong typing**.

11.3.1 Generic Interfaces

A generic interface can represent a structure that works with **any type** specified by the user.

Example: `RepositoryT` Interface

```
interface Repository<T> {
  getById(id: number): T;
  getAll(): T[];
  save(item: T): void;
}
```

You can now implement this interface for different data models:

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

class UserRepository implements Repository<User> {
  private users: User[] = [];

  getById(id: number): User {
    return this.users.find(user => user.id === id)!;
  }

  getAll(): User[] {
    return this.users;
  }

  save(user: User): void {
    this.users.push(user);
  }
}
```

By specifying `User` as the generic type `T`, the `UserRepository` enforces that all operations are type-safe for the `User` type.

11.3.2 Generic Classes

Generic classes are useful when you want to create a reusable data structure that can work with multiple types.

Example: `BoxT` Class

Full runnable code:

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

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

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

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

const numberBox = new Box<number>(123);
console.log(numberBox.getValue()); // Output: 123

const stringBox = new Box<string>("hello");
console.log(stringBox.getValue()); // Output: hello
```

This same `Box<T>` class works with any type while preserving type safety.

11.3.3 Why Use Generics with Interfaces and Classes?

Benefit	Description
Flexibility	Write one definition that works with many types
Reusability	Avoid duplicating logic for different data shapes
Type Safety	TypeScript ensures correct usage of the provided type
Better Tooling	Autocompletion and type checking in IDEs

11.3.4 Summary

- **Generic interfaces** define contracts for a family of types (e.g., `Repository<T>`).
- **Generic classes** allow reusable logic over different data types (e.g., `Box<T>`).
- Generics increase **code reusability** and **type safety** at the same time.
- You specify the type when using the interface/class to "fill in the blank" for `T`.

Using generics with interfaces and classes is a core part of building scalable, type-safe TypeScript applications.

11.4 Real-World Generic Examples

Generics are especially powerful when writing reusable components and utility types in real-world applications. In this section, we'll walk through several **compact and practical examples** that demonstrate how generics help create flexible, type-safe solutions.

11.4.1 Example 1: Dropdown Component

A dropdown that works with any item type (`string`, `number`, or custom object).

Full runnable code:

```
interface DropdownOption<T> {
  label: string;
  value: T;
}

function renderDropdown<T>(options: DropdownOption<T>[]): void {
  options.forEach(option => {
    console.log(`Label: ${option.label}, Value: ${option.value}`);
  });
}

// Usage with strings
renderDropdown<string>([
  { label: "Apple", value: "apple" },
  { label: "Banana", value: "banana" },
]);

// Usage with numbers
renderDropdown<number>([
  { label: "One", value: 1 },
  { label: "Two", value: 2 },
]);
```

YES This makes the component reusable for any data type.

11.4.2 Example 2: Paginated API Response

Model an API response that wraps a list of results and pagination info.

Full runnable code:

```
interface PaginatedResponse<T> {
  data: T[];
  currentPage: number;
  totalPages: number;
}
```

```
// Usage with a Product type
interface Product {
  id: number;
  name: string;
}

const productResponse: PaginatedResponse<Product> = {
  data: [
    { id: 1, name: "Laptop" },
    { id: 2, name: "Tablet" },
  ],
  currentPage: 1,
  totalPages: 3,
};

console.log(productResponse.data[0].name); // Output: Laptop
```

YES This allows consistent pagination while keeping the result type generic.

11.4.3 Example 3: Queue Class

A generic queue that supports enqueue and dequeue operations for any item type.

Full runnable code:

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

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

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

  peek(): T | undefined {
    return this.items[0];
  }
}

// Usage with strings
const messageQueue = new Queue<string>();
messageQueue.enqueue("First");
messageQueue.enqueue("Second");
console.log(messageQueue.dequeue()); // Output: First

// Usage with numbers
const numberQueue = new Queue<number>();
numberQueue.enqueue(100);
console.log(numberQueue.peek()); // Output: 100
```

YES This queue works seamlessly for any type with full type safety.

11.4.4 Summary

Example	Use Case
Dropdown Component	Flexible UI options
Paginated API Response	Reusable API model for multiple entities
Generic Queue	Data structure usable across all types

Generics make it easy to build reusable, scalable components and data models without sacrificing **type safety**, which is especially valuable in real-world TypeScript development.

Chapter 12.

Modules and Namespaces

1. Import and Export Syntax
2. Organizing Code with Modules
3. Legacy Namespaces and When to Use Them

12 Modules and Namespaces

12.1 Import and Export Syntax

One of TypeScript's most powerful features is its support for **modular programming** using **ES module syntax**. Modules allow you to split your code into separate files and reuse logic across your application in a clean, maintainable way.

TypeScript uses the same **`import`** and **`export`** syntax as modern JavaScript (ES6+), with full type support.

12.1.1 Exporting Code from a Module

There are **two main ways** to export things in TypeScript:

Named Exports

Use `export` in front of functions, variables, classes, or interfaces:

```
// mathUtils.ts
export const PI = 3.14159;

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

export class Circle {
  constructor(public radius: number) {}
}
```

You can export multiple named items from the same file.

Default Export

Use `export default` when you want to export a **single main value** from a file:

```
// greeting.ts
export default function greet(name: string): string {
  return `Hello, ${name}!`;
}
```

A module can have **only one default export**, but can still include named exports too (not recommended for beginners as it can get confusing).

12.1.2 Importing Modules

Import Named Exports

Use curly braces `{}` to import named exports:

```
// main.ts
import { PI, add, Circle } from './mathUtils';

console.log(add(2, 3));         // Output: 5
console.log(PI);                // Output: 3.14159
const c = new Circle(5);
```

You can also rename imports:

```
import { add as sum } from './mathUtils';
console.log(sum(1, 2)); // Output: 3
```

Import Default Export

Use any name you want when importing a default export:

```
import greet from './greeting';

console.log(greet("Alice")); // Output: Hello, Alice!
```

You don't use `{}` for default imports.

Mixed Export Example (Not Recommended for Beginners)

```
// mixedExample.ts
export const version = "1.0";
export default function hello() {
  console.log("Hello from default");
}
```

```
// main.ts
import hello, { version } from './mixedExample';
hello();           // Output: Hello from default
console.log(version); // Output: 1.0
```

Although valid, mixing default and named exports can reduce clarity, especially for new learners.

12.1.3 Example Project Structure

```
/project
  +-- main.ts
  +-- mathUtils.ts
  +-- greeting.ts
```

- `mathUtils.ts` contains named exports (`add`, `PI`, `Circle`)

- `greeting.ts` has a default export (`greet`)
- `main.ts` imports from both modules

12.1.4 Summary: Named vs Default Exports

Feature	Named Export	Default Export
Syntax	`export function foo() {}`	`export default function() {}`
Import Style	`import { foo } from './x'`	`import foo from './x'`
Multiple per file	YES Yes	NO No
Common Use Case	Utility libraries	Single main export per file

TypeScript's module system helps keep your code **organized, reusable, and easy to understand**. Learning how to use `import` and `export` correctly is key to managing large-scale TypeScript projects.

12.2 Organizing Code with Modules

As TypeScript applications grow, organizing your code into **modules** (separate files) becomes essential for **readability**, **reusability**, and **maintainability**. This section shows how to structure your project using modules and how to configure TypeScript to support them.

12.2.1 Why Use Modules?

Modules help you:

- **Break large codebases into manageable parts**
- **Reuse code across files**
- **Avoid global variable pollution**
- **Control what is publicly available (exports)**

12.2.2 Example Project Structure

Here's a simple project that contains utility functions, an interface, a class, and a main file to run everything:

```
/project
  +-- src
      +-- main.ts
      +-- mathUtils.ts
      +-- logger.ts
      +-- models
          +-- Product.ts
  +-- tsconfig.json
```

12.2.3 File: mathUtils.ts

```ts
// src/mathUtils.ts
export function add(a: number, b: number): number {
  return a + b;
}

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

12.2.4 File: logger.ts

```ts
// src/logger.ts
export function log(message: string): void {
  console.log(`[LOG]: ${message}`);
}
```

12.2.5 File: models/Product.ts

```ts
// src/models/Product.ts

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

export class ProductService {
  private products: Product[] = [];

  addProduct(p: Product): void {
    this.products.push(p);
  }
```

```
  getAll(): Product[] {
    return this.products;
  }
}
```

12.2.6 File: `main.ts`

```
// src/main.ts
import { add, multiply } from './mathUtils';
import { log } from './logger';
import { ProductService, Product } from './models/Product';

const total = add(10, 5);
log(`Total: ${total}`);

const productService = new ProductService();

const newProduct: Product = { id: 1, name: "Phone", price: 599 };
productService.addProduct(newProduct);

log(`Products: ${JSON.stringify(productService.getAll())}`);
```

12.2.7 Configuring `tsconfig.json`

To make sure TypeScript understands your module structure, you need a basic `tsconfig.json` in your root folder:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "es6",
    "moduleResolution": "node",
    "rootDir": "./src",
    "outDir": "./dist",
    "strict": true,
    "esModuleInterop": true
  },
  "include": ["src"]
}
```

Key Options:

Option	Description
`module`	Determines module system (e.g. `commonjs`, `es6`)
`moduleResolution`	Helps locate modules in folders like `node_modules`
`rootDir`	Where source files live (`src`)

Option	Description
`outDir`	Where compiled JS goes (`dist`)

12.2.8 Compiling the Project

Run the TypeScript compiler:

```
npx tsc
```

This compiles all files in `src/` into JavaScript files inside the `dist/` folder.

12.2.9 Benefits of Modular Code

- **Improved readability:** Each file has a single responsibility.
- **Scalability:** Easy to add or refactor features.
- **Reusability:** Utility functions and models can be reused across projects.
- **Testability:** Smaller units are easier to test individually.

12.2.10 Summary

Organizing your TypeScript code using **modules and folders** helps manage growing codebases. With `import` and `export`, you can keep your logic clean and maintainable. Setting up `tsconfig.json` ensures TypeScript compiles your modules correctly.

In the next section, we'll look at **namespaces** and when they might still be useful.

12.3 Legacy Namespaces and When to Use Them

Before modern JavaScript added native module support (via `import/export`), TypeScript provided its own way of organizing code: the `namespace`. Namespaces allowed developers to group related variables, functions, interfaces, and classes under a single name, helping avoid global scope pollution.

Although **modules are now the preferred way** to structure TypeScript code, understanding namespaces is useful for working with legacy code or environments where modules aren't available (such as some browser scripts or embedded environments).

12.3.1 What Is a `namespace`?

A `namespace` is a way to logically group code under a named wrapper. Everything inside a namespace is kept off the global scope unless explicitly exported.

Example: Simple Namespace

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

  export function subtract(a: number, b: number): number {
    return a - b;
  }
}

console.log(MathUtils.add(5, 3));      // Output: 8
console.log(MathUtils.subtract(5, 3)); // Output: 2
```

Here, `MathUtils` is a namespace that groups math-related functions. You must **prefix** access with the namespace name (`MathUtils.add`).

12.3.2 Nested Namespaces

Namespaces can be **nested** to further organize code.

```
namespace App {
  export namespace Models {
    export interface User {
      id: number;
      name: string;
    }
  }

  export namespace Services {
    export function greet(user: Models.User): string {
      return `Hello, ${user.name}`;
    }
  }
}

const user: App.Models.User = { id: 1, name: "Alice" };
console.log(App.Services.greet(user)); // Output: Hello, Alice
```

12.3.3 Differences: Namespaces vs Modules

Feature	`namespace`	`module` (ES modules)
Scope	Global (unless bundled)	File-based (each file is a module)
Syntax	`namespace Name {}`	`export` / `import`
Compilation	Works without module support	Requires bundler or module-aware runtime
Target	Legacy or in-browser scripts	Modern TypeScript projects
Usage Today		

12.3.4 When to Use Namespaces

Use namespaces only if:

- You are writing TypeScript to be used **in-browser directly via `<script>` tags**.
- You are maintaining **legacy code** that already uses namespaces.
- You are building a **library** to be included without a module system (e.g., in a CDN-delivered global script).

Do NOT use namespaces if:

- You are using modern build tools (e.g., Webpack, Vite, or TypeScript projects with `import`/`export`).
- You are building a modular, large-scale application.

12.3.5 Summary

- `namespace` is a legacy feature used to group related declarations in the global scope.
- Today, **ES modules are the preferred approach** in TypeScript projects.
- Namespaces can still be useful in **non-module environments**, such as browser scripts that load TypeScript via `<script>` tags.

Unless you're targeting a special use case, stick with `import`/`export` modules for better maintainability and compatibility with modern tooling.

Chapter 13.

Asynchronous Programming in TypeScript

1. Promises and `async/await`
2. Typing Asynchronous Functions
3. Fetching Data (e.g., with `fetch`)
4. Handling Errors in Async Code

13 Asynchronous Programming in TypeScript

13.1 Promises and `async/await`

Modern applications often need to perform **asynchronous operations**—tasks that take time to complete, such as fetching data from a server or reading a file. TypeScript uses **Promises** and the `async/await` syntax to make handling such operations clean and predictable.

13.1.1 What is a Promise?

A `Promise` represents a value that may be **available now**, **in the future**, or **never**. It allows you to register callbacks to handle the result when it's ready.

Promise States

A Promise has three states:

State	Description
`pending`	Initial state, operation not completed yet
`fulfilled`	Operation completed successfully (`resolve`)
`rejected`	Operation failed (`reject`)

13.1.2 Creating and Using a Promise

Here's a basic example of creating a Promise that resolves after a delay:

```
function delay(ms: number): Promise<string> {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Finished after ${ms}ms`);
    }, ms);
  });
}
```

You can consume this promise using `.then()` and `.catch()`:

```
delay(1000)
  .then((message) => {
    console.log("Resolved:", message);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

13.1.3 `async` and `await`: Cleaner Syntax

The `async`/`await` syntax provides a way to **write asynchronous code that looks synchronous**, improving readability and maintainability.

Example: Refactored with `async`/`await`

Full runnable code:

```
async function runDelay() {
  try {
    const result = await delay(1000);
    console.log("Resolved:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}

runDelay();
```

Explanation:

- `async` makes the function return a Promise.
- `await` pauses execution until the Promise resolves or rejects.
- `try/catch` is used for error handling, just like in synchronous code.

13.1.4 Example: Simulated Data Fetch

Full runnable code:

```
function fetchData(): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data loaded"), 1500);
  });
}

async function load() {
  const data = await fetchData();
  console.log(data);
}

load(); // Output after 1.5s: Data loaded
```

This simple pattern—**define a Promise-based function, then use `await` in an `async` function**—is at the heart of modern asynchronous programming in TypeScript.

13.1.5 Benefits of `async/await`

- YES **Improved readability** over `.then()` chains
- YES **Synchronous-like flow** for asynchronous logic
- YES **Easier error handling** with `try/catch`

13.1.6 Summary

Concept	Use
`Promise`	Represents a future result
`.then()` / `.catch()`	Traditional syntax for handling results
`async/await`	Cleaner, more readable way to work with Promises

13.2 Typing Asynchronous Functions

TypeScript makes asynchronous code safer and more maintainable by allowing you to annotate the return types of functions that use Promises. Explicitly specifying the return type of an `async` function improves **type safety**, **code readability**, and **developer tooling support** (like auto-complete and linting).

13.2.1 Annotating Async Function Return Types

When you define an `async` function, TypeScript infers the return type to be `Promise<T>`, where `T` is the value you `return`.

Example: Inferred Return Type

```
async function greet(): Promise<string> {
  return "Hello";
}
```

Even without writing `: Promise<string>`, TypeScript will infer it—but **explicit annotations** make your code clearer and easier to maintain.

13.2.2 Typing with `PromiseT`

The generic type `Promise<T>` represents a Promise that resolves to a value of type `T`.

Example: Async Function Returning a Number

```
async function getUserAge(): Promise<number> {
  return 30;
}
```

TypeScript ensures the return value is a `number`. If you accidentally return a `string`, it will show a type error.

13.2.3 Typing a Promise-Based Function (Not Using `async`)

You can also return Promises manually using the `new Promise` constructor:

```
function fetchMessage(): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Done!"), 1000);
  });
}
```

This is equivalent in type to:

```
async function fetchMessage(): Promise<string> {
  return "Done!";
}
```

13.2.4 Typing with Generics

For reusable async functions, you can use **generic types**:

Example: Generic Fetch Wrapper

```
async function fetchData<T>(url: string): Promise<T> {
  const response = await fetch(url);
  if (!response.ok) {
    throw new Error("Failed to fetch");
  }
  return await response.json() as T;
}
```

You can then use it like this:

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

const userUrl = "https://api.example.com/user/1";
```

```ts
async function loadUser() {
  const user = await fetchData<User>(userUrl);
  console.log(user.name); // Safe: name is typed as string
}
```

Benefits of Generic Typing:

- YES Flexible: Reuse across multiple data shapes
- YES Safe: TypeScript enforces structure of fetched data
- YES Intuitive: Code editors can suggest property names

13.2.5 Optional: Combining With Interfaces

```ts
interface ApiResponse<T> {
  success: boolean;
  data: T;
}

async function getApiResponse<T>(): Promise<ApiResponse<T>> {
  return {
    success: true,
    data: {} as T,
  };
}
```

13.2.6 Why Use Explicit Types?

- **Prevent bugs**: Type mismatches are caught during development
- **Improve tooling**: Editors offer better auto-completion and refactoring support
- **Document intent**: Return types clarify what a function is supposed to produce

13.2.7 Summary

Pattern	Example
Basic async return	`async function getValue(): Promise<number> { ... }`
Manual Promise	`function getValue(): Promise<number> { return Promise.resolve(42); }`
With generics	`async function load<T>(): Promise<T> { ... }`

By explicitly typing your asynchronous functions with `Promise<T>`, you make your codebase

safer, cleaner, and easier to understand—especially in larger projects or when collaborating with others.

13.3 Fetching Data (e.g., with `fetch`)

Fetching data from an API is a common task in any application. TypeScript adds strong typing to the `fetch` API, helping you safely parse and use JSON responses.

13.3.1 Basic `fetch` in TypeScript

The `fetch()` function returns a `Promise<Response>`, which you usually convert to JSON:

```
fetch("https://api.example.com/users")
  .then((res) => res.json())
  .then((data) => console.log(data));
```

However, without type annotations, this data is treated as `any`, which can lead to **runtime errors**. TypeScript helps by letting you **define the expected shape** of the response.

13.3.2 Typing the Response

Step 1: Define an Interface for Your Data

Suppose we are calling an API that returns a user:

```
{
  "id": 1,
  "name": "Alice",
  "email": "alice@example.com"
}
```

We can define a corresponding interface in TypeScript:

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

Step 2: Use `fetch` with Type Assertion

```
async function getUser(id: number): Promise<User> {
  const res = await fetch(`https://api.example.com/users/${id}`);

  if (!res.ok) {
```

```
    throw new Error("Failed to fetch user");
  }

  const data = await res.json();
  return data as User;
}
```

Here, we **assert** that `data` is of type `User` using `as User`.

YES **Pros**: Clear structure WARNING **Caution**: Type assertion (`as User`) doesn't actually check the shape—it just tells TypeScript to trust you.

13.3.3 Safer Parsing with Runtime Validation (Bonus Tip)

If you need **runtime safety**, consider validating the structure manually or using libraries like `zod` or `io-ts`.

Manual check:

```
function isUser(obj: any): obj is User {
  return typeof obj.id === "number" &&
         typeof obj.name === "string" &&
         typeof obj.email === "string";
}
```

Then use:

```
if (!isUser(data)) {
  throw new Error("Invalid user data");
}
```

13.3.4 Complete Example: Typed Fetch

Full runnable code:

```
interface Post {
  userId: number;
  id: number;
  title: string;
  body: string;
}

async function fetchPost(postId: number): Promise<Post> {
  const response = await fetch(`https://jsonplaceholder.typicode.com/posts/${postId}`);

  if (!response.ok) {
    throw new Error("Network response was not ok");
  }
```

```
  const data = await response.json();
  return data as Post;
}

fetchPost(1)
  .then((post) => console.log("Post title:", post.title))
  .catch((error) => console.error("Error fetching post:", error));
```

13.3.5 Common Pitfalls to Avoid

Pitfall	Solution
Blindly using `any`	Define interfaces or types
Not checking `res.ok`	Always check `response.ok` before parsing
Assuming JSON shape	Use type assertions or runtime validation
Ignoring async errors	Wrap with `try/catch` or use `.catch()`

13.3.6 Summary

- Use `fetch` with async/await for clean syntax.
- Define interfaces to represent expected response data.
- Assert or validate JSON responses to help TypeScript catch errors early.
- Use `response.ok` to handle failed HTTP responses.

By combining TypeScript with `fetch`, you gain strong type checking and better developer tooling—making your code safer and more maintainable.

13.4 Handling Errors in Async Code

When working with asynchronous operations in TypeScript, error handling is crucial to ensure your application behaves predictably—even when something goes wrong. Whether you're fetching data from an API or reading a file, handling errors properly improves reliability and user experience.

13.4.1 Using `.catch()` with Promises

For Promise-based code, you can attach a `.catch()` method to handle errors:

```
fetch("https://api.example.com/data")
  .then((res) => res.json())
  .then((data) => {
    console.log("Data received:", data);
  })
  .catch((error) => {
    console.error("Fetch failed:", error);
  });
```

Note: `.catch()` will handle both network errors and exceptions thrown in previous `.then()` blocks.

13.4.2 Using `try/catch` with `async/await`

With `async/await`, error handling becomes more readable and structured using `try/catch`:

Full runnable code:

```
async function loadData() {
  try {
    const res = await fetch("https://api.example.com/data");

    if (!res.ok) {
      throw new Error(`HTTP error! status: ${res.status}`);
    }

    const data = await res.json();
    console.log("Data loaded:", data);
  } catch (error) {
    console.error("Error loading data:", error);
  }
}

loadData();
```

YES **Benefits**:

- Linear, readable code
- Clear error boundaries
- Easier to reason about logic and fallback behavior

13.4.3 Handling HTTP Errors Gracefully

Not all API errors throw exceptions—some return valid responses with error status codes. TypeScript helps ensure you check these:

```
async function getUser(id: number): Promise<void> {
  try {
```

```
    const response = await fetch(`https://api.example.com/users/${id}`);

    if (!response.ok) {
      // 4xx or 5xx error
      console.warn(`Request failed with status: ${response.status}`);
      return;
    }

    const user = await response.json();
    console.log("User data:", user);
  } catch (err) {
    console.error("Network or parsing error:", err);
  }
}
```

13.4.4 Distinguishing Error Types with TypeScript

TypeScript knows little about the shape of an error—so we often need to **narrow the type** manually:

```
try {
  // some async code
} catch (err) {
  if (err instanceof Error) {
    console.error("Message:", err.message);
  } else {
    console.error("Unknown error:", err);
  }
}
```

This guards against non-`Error` values being thrown (e.g., `throw "oops"`), helping maintain type safety.

13.4.5 Common Best Practices

Practice	Description
YES Check `response.ok`	Never assume fetch succeeded—always validate response status
YES Use `try/catch`	Preferred with `async/await` for clean, scoped error handling
YES Handle specific errors	Use `instanceof` or error codes to tailor the response
WARNING Avoid silent failures	Always log or respond to errors so they don't go unnoticed
YES Provide fallback logic	Allow your app to recover gracefully (e.g., show a message to the user)

13.4.6 Complete Example

```
async function fetchPost(id: number): Promise<void> {
  try {
    const res = await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`);

    if (!res.ok) {
      throw new Error(`Failed to fetch: ${res.status}`);
    }

    const post = await res.json();
    console.log("Post:", post);
  } catch (error) {
    if (error instanceof Error) {
      console.error("Fetch error:", error.message);
    } else {
      console.error("Unknown error:", error);
    }
  }
}

fetchPost(1);
```

13.4.7 Summary

- Use `.catch()` with Promises and `try/catch` with `async/await` for clean, predictable error handling.
- Always validate HTTP responses (e.g., with `response.ok`).
- Leverage TypeScript's type system to handle known and unknown errors safely.
- Writing clear and consistent error-handling logic makes your code more robust and user-friendly.

Chapter 14.

Advanced Type Manipulation

1. Mapped Types
2. Conditional Types
3. Indexed Access and Lookup Types
4. Template Literal Types

14 Advanced Type Manipulation

14.1 Mapped Types

Mapped types in TypeScript allow you to create new types by transforming properties of existing types. This is especially useful when you want to reuse or modify a structure without repeating code.

At the core of mapped types is the `in` keyword, which lets you iterate over keys in a type.

14.1.1 Syntax

```
type NewType = {
  [Key in keyof ExistingType]: Type
};
```

- `keyof ExistingType`: gets all keys of the type as a union.
- `Key in keyof ExistingType`: loops over those keys.
- The mapped type modifies each property as specified.

14.1.2 Built-In Mapped Types

`PartialT`

Makes all properties in `T` optional.

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

type PartialUser = Partial<User>;

// Equivalent to:
type PartialUserManual = {
  id?: number;
  name?: string;
};
```

`ReadonlyT`

Makes all properties in `T` read-only.

```
type ReadonlyUser = Readonly<User>;

// Equivalent to:
type ReadonlyUserManual = {
  readonly id: number;
```

```
  readonly name: string;
};
```

14.1.3 Creating Custom Mapped Types

You can create your own reusable mapped types based on what you want to achieve.

Example: Convert all properties to `boolean`

```
type Flags<T> = {
  [K in keyof T]: boolean;
};

type FeatureToggles = Flags<{
  darkMode: () => void;
  newDashboard: () => void;
}>;

// Result:
type FeatureToggles = {
  darkMode: boolean;
  newDashboard: boolean;
}
```

Example: Remove `readonly` and make everything mutable

```
type Mutable<T> = {
  -readonly [K in keyof T]: T[K];
};

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

type MutablePerson = Mutable<ReadonlyPerson>;
// name and age are now mutable
```

The `-readonly` syntax removes `readonly` from each property.

14.1.4 Practical Use Case

Mapped types help create flexible APIs and reduce duplication. For instance, if you're building a form where fields may be partially filled:

```
interface Product {
  id: number;
  title: string;
```

```
  price: number;
}

function updateProduct(id: number, fieldsToUpdate: Partial<Product>) {
  // Logic to update only specific fields
}
```

This function allows updating only some fields of a `Product` without creating a new custom type manually.

14.1.5 Summary

Feature	Benefit
`Partial<T>`	Makes all properties optional
`Readonly<T>`	Makes all properties read-only
Custom mapped types	Create flexible, reusable type tools

Mapped types are a powerful feature that improve type reusability and safety, especially in large codebases where object shapes evolve or are reused across different contexts.

14.2 Conditional Types

Conditional types in TypeScript work like type-level `if-else` statements. They allow you to select one type or another based on a condition evaluated at compile time. This powerful feature helps you create flexible and precise types that depend on input types.

14.2.1 Syntax

```
T extends U ? X : Y
```

- `T`: The type to check.
- `U`: The type against which `T` is tested.
- `X`: The resulting type if `T` is assignable to `U`.
- `Y`: The resulting type if `T` is not assignable to `U`.

This reads as: *If `T` is assignable to `U`, then use type `X`, else use type `Y`.*

14.2.2 Practical Examples

Example 1: Differentiate Arrays vs Non-Arrays

```
type IsArray<T> = T extends any[] ? "Yes, array" : "No, not array";

type Test1 = IsArray<string[]>;   // "Yes, array"
type Test2 = IsArray<number>;     // "No, not array"
```

Here, the type `IsArray<T>` checks if `T` extends `any[]` (i.e., is an array type). It returns different string literals accordingly.

Example 2: Extracting Promise Resolved Type

When working with asynchronous code, you may want a type helper that extracts the type a `Promise` resolves to.

```
type UnwrapPromise<T> = T extends Promise<infer U> ? U : T;

type Result1 = UnwrapPromise<Promise<string>>;  // string
type Result2 = UnwrapPromise<number>;           // number
```

- Here, `infer U` allows capturing the type inside the `Promise`.
- If `T` is a `Promise<U>`, `UnwrapPromise<T>` resolves to `U`.
- Otherwise, it just returns `T`.

Example 3: Filter Out Functions from a Union

```
type NonFunction<T> = T extends Function ? never : T;

type Mixed = string | (() => void) | number;

type Filtered = NonFunction<Mixed>;  // string | number
```

- `NonFunction<T>` removes any function types from the union by replacing them with `never`.
- This is useful to filter or narrow types.

14.2.3 Why Use Conditional Types?

- They allow building **type utilities** that adapt based on input types.
- Help make your types **more expressive and safe**.
- Enable **type inference** that mirrors complex runtime logic.

14.2.4 Summary

Concept	Description
`T extends U ? X : Y`	Type-level conditional branching
`infer` keyword	Extracts types within other types
Use cases	Differentiate arrays, unwrap promises, filter unions

Conditional types are a cornerstone of advanced TypeScript typing. Mastering them unlocks the ability to write smarter, more maintainable, and type-safe code.

14.3 Indexed Access and Lookup Types

TypeScript's **indexed access types** (also called *lookup types*) let you retrieve the type of a property from an object type dynamically. This powerful feature enables you to access property types programmatically, making your types more flexible and reusable.

14.3.1 Syntax

```
T[K]
```

- `T` is an object type.
- `K` is a key (or union of keys) of `T`.
- The expression `T[K]` evaluates to the type of the property `K` on `T`.

14.3.2 Basic Example: Extracting a Property Type

```ts
interface Person {
  name: string;
  age: number;
  isAdmin: boolean;
}

// Get the type of the 'name' property
type NameType = Person["name"];  // string

// Get the type of the 'age' property
type AgeType = Person["age"];    // number
```

Here, `Person["name"]` extracts the type of the `name` property (`string`), and `Person["age"]` extracts `number`.

14.3.3 Using Union of Keys

You can use a union of keys to get a union of property types:

```
type NameOrAdmin = Person["name" | "isAdmin"];  // string | boolean
```

This means `NameOrAdmin` can be either `string` or `boolean`.

14.3.4 Dynamic Access Using `keyof`

You can combine indexed access with the `keyof` operator to write generic utilities:

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

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

const personName = getProperty(person, "name");    // Type inferred as string
const personAge = getProperty(person, "age");      // Type inferred as number
```

- The function `getProperty` accepts an object and a key.
- It returns the property value with the exact type `T[K]`.
- TypeScript enforces that `key` must be a valid key of `T`.

14.3.5 Practical Utility Type: Property Types of Object

You can create reusable type utilities that operate on object keys:

```
// Get all property types of T as a union
type PropertyTypes<T> = T[keyof T];

type PersonPropTypes = PropertyTypes<Person>;  // string | number | boolean
```

14.3.6 Why Use Indexed Access Types?

- They enable **type-safe dynamic access** to properties.
- Support **generic programming** where property names or keys may vary.
- Help build **advanced utilities** and **conditional logic** on types.

14.3.7 Summary

Concept	Description
`T[K]`	Lookup type: type of property `K` on type `T`
`keyof T`	Union of keys of type `T`
Combining `T[K]` and `keyof T`	Generic, type-safe access to object properties

Indexed access types let TypeScript programmatically work with object shapes and make your code more flexible while maintaining strict type safety.

14.4 Template Literal Types

TypeScript's **template literal types** allow you to create new string literal types by combining unions of strings using template literal syntax. This feature brings the power of template literals from runtime to the type system, enabling **flexible**, **type-safe string manipulations**.

14.4.1 What Are Template Literal Types?

Template literal types build new string types by embedding unions of strings into template string patterns, similar to JavaScript template literals but evaluated at the type level.

14.4.2 Basic Syntax

```
type Greeting = `Hello, ${'Alice' | 'Bob' | 'Carol'}`;
```

Here, `Greeting` is a union of the string literals:

- `"Hello, Alice"`
- `"Hello, Bob"`
- `"Hello, Carol"`

14.4.3 Examples

Prefixing and Suffixing String Unions

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

// Add a prefix
```

```
type PrefixedSizes = `size-${Sizes}`;
// Result: "size-small" | "size-medium" | "size-large"

// Add a suffix
type CssClasses = `btn-${Sizes}`;
// Result: "btn-small" | "btn-medium" | "btn-large"
```

This technique helps enforce consistent string formats across your codebase.

Combining Literal Types for Dynamic Keys

Template literal types are useful for building dynamic keys or event names:

```
type EventNames = "click" | "focus" | "blur";

type PrefixedEvents = `on${Capitalize<EventNames>}`;
// Result: "onClick" | "onFocus" | "onBlur"
```

Here, we used the built-in utility type `Capitalize<>` to capitalize the first letter of each event name.

14.4.4 Real-World Use Case: API Endpoint Paths

Imagine an API with different resource types and actions:

```
type Resource = "user" | "post" | "comment";
type Action = "create" | "delete" | "update";

type Endpoint = `/api/${Resource}/${Action}`;

// Examples of Endpoint values:
// "/api/user/create"
// "/api/post/delete"
// "/api/comment/update"
```

Using template literal types like this helps TypeScript **enforce valid URL strings** in your API calls.

14.4.5 Strongly Typed String Patterns

Template literal types can model more complex string patterns, enabling safer string processing in APIs or libraries:

```
type CssProperties = "width" | "height" | "color";

type CssPropertyWithUnit = `${CssProperties}-${"px" | "em" | "%"}`;
// Possible values:
// "width-px" | "width-em" | "width-%" | "height-px" | ...
```

14.4.6 Summary

Feature	Description
Template literal types	Combine unions of string literals with template strings
Use cases	Dynamic keys, API paths, event names, CSS classes
Type utilities integration	Works with `Capitalize<>`, `Uppercase<>`, and more

Template literal types extend TypeScript's type system to handle dynamic strings with full type safety, making your code more robust and expressive.

Chapter 15.

Decorators and Metadata

1. Introduction to Decorators
2. Class, Method, and Property Decorators
3. Use Cases: Logging, Validation, Metadata

15 Decorators and Metadata

15.1 Introduction to Decorators

Decorators are a powerful and expressive feature in TypeScript that allow you to **modify or enhance classes, methods, properties, or parameters** at design time, using special functions called *decorators*. Think of decorators as wrappers or annotations that add metadata or alter behavior without changing the original code structure directly.

15.1.1 What Are Decorators?

A **decorator** is essentially a function that is applied to a target — such as a class or method — to observe, modify, or replace it. This allows developers to implement cross-cutting concerns like logging, validation, or dependency injection in a clean and reusable way.

Why Decorators Matter

- **Separation of concerns:** Decorators let you separate auxiliary logic (e.g., logging) from business logic.
- **Reusable enhancements:** You can apply the same decorator to many classes or methods without repeating code.
- **Metadata and reflection:** Decorators can attach metadata to elements, enabling frameworks to use reflection for advanced features.

15.1.2 Decorator Syntax

Decorators use the `@` symbol followed by the decorator function name, placed immediately before the declaration they modify:

```
@sealed
class MyClass {
  // ...
}

function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}
```

Here, `@sealed` is a class decorator that seals the class constructor and its prototype, preventing further modification.

15.1.3 What Can Be Decorated?

TypeScript supports decorators for:

- **Classes:** Modify or replace class constructors.
- **Methods:** Alter method behavior or metadata.
- **Properties:** Intercept or modify property definitions.
- **Parameters:** Add metadata to method parameters.

15.1.4 Status of Decorators in JavaScript and TypeScript

- **JavaScript:** Decorators are a **stage 3 proposal** in the ECMAScript standardization process, meaning the specification is nearing completion but not finalized.
- **TypeScript:** Supports decorators as an **experimental feature**, enabled via the `experimentalDecorators` compiler option in `tsconfig.json`. This allows you to write decorator code today while anticipating future standardization.

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

15.1.5 Summary

Aspect	Explanation
What are decorators?	Functions that modify classes, methods, or properties at design time
Syntax	Use `@decoratorName` before declarations
Use cases	Logging, validation, metadata, dependency injection
JavaScript status	Stage 3 proposal — nearly standardized
TypeScript support	Experimental, opt-in via `experimentalDecorators` flag

Decorators unlock advanced programming patterns in TypeScript, offering flexible ways to extend and control your code's behavior with clean, declarative syntax.

15.2 Class, Method, and Property Decorators

In TypeScript, decorators come in different flavors depending on what they modify: **classes, methods, or properties**. Each type has a different signature and use case. Below, we'll explore how to create and apply each kind.

15.2.1 Class Decorators

Class decorators receive the **constructor function** of the class they decorate. You can modify the class, replace it, or add static properties.

Example: A class decorator that logs when the class is created

```
// A simple class decorator that logs when a class is instantiated
function Logger(constructor: Function) {
  const original = constructor;

  function newConstructor(...args: any[]) {
    console.log(`Creating instance of ${original.name}`);
    return new original(...args);
  }

  // Copy prototype so instanceof works correctly
  newConstructor.prototype = original.prototype;

  return newConstructor as any;
}

@Logger
class Person {
  constructor(public name: string) {}
}

const p = new Person("Alice");
// Console output: Creating instance of Person
```

Explanation:

- The decorator `Logger` wraps the original constructor, logging on each new instance creation.
- Returning a new constructor replaces the original class.

15.2.2 Method Decorators

Method decorators receive:

- The **target** (prototype or constructor)

- The **method name**
- The **property descriptor** to modify the method behavior

Example: A method decorator that logs method calls and arguments

```
function LogMethod(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  const originalMethod = descriptor.value;

  descriptor.value = function (...args: any[]) {
    console.log(`Called ${propertyKey} with args:`, args);
    const result = originalMethod.apply(this, args);
    console.log(`Returned:`, result);
    return result;
  };
}

class Calculator {
  @LogMethod
  add(a: number, b: number): number {
    return a + b;
  }
}

const calc = new Calculator();
calc.add(3, 5);
// Console output:
// Called add with args: [3, 5]
// Returned: 8
```

Explanation:

- The decorator wraps the original method to log input arguments and the result.
- It modifies the method's `descriptor.value`.

15.2.3 Property Decorators

Property decorators receive:

- The **target** (prototype or constructor)
- The **property name**

Note that property decorators **do not provide a property descriptor**, so they can only add metadata or perform initialization side effects.

Example: A property decorator that marks a property as required (for demonstration)

Full runnable code:

```
function Required(target: any, propertyKey: string) {
  // Store metadata about required fields (simplified example)
  if (!target.constructor.requiredProps) {
    target.constructor.requiredProps = [];
  }
  target.constructor.requiredProps.push(propertyKey);
}

class User {
  @Required
  username!: string;

  @Required
  email!: string;

  constructor(username: string, email: string) {
    this.username = username;
    this.email = email;
  }
}

// Later, you could check for required fields:
console.log(User.requiredProps); // ['username', 'email']
```

Explanation:

- This decorator adds metadata about which properties are required.
- It doesn't enforce validation by itself but can be used by other parts of the program.

15.2.4 Summary

Decorator Type	Signature	Purpose	Example Use Case
Class	`(constructor: Function) => Function | void`	Modify or replace the class constructor	Logging, dependency injection
Method	`(target, propertyKey, descriptor) => void`	Wrap or modify method behavior	Logging calls, timing
Property	`(target, propertyKey) => void`	Attach metadata or validation	Required fields, serialization

Decorators provide a declarative and powerful way to extend and customize classes, methods, and properties, enabling clean and reusable design patterns.

15.3 Use Cases: Logging, Validation, Metadata

Decorators provide a clean, reusable way to add extra behavior or metadata to classes, methods, and properties without cluttering the main logic. Below are some practical real-world examples demonstrating their power.

15.3.1 Logging Method Calls and Arguments Automatically

Instead of adding manual `console.log()` statements inside every method, you can write a **method decorator** that logs calls, arguments, and results automatically.

```
function LogCall(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  const originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${propertyKey} with arguments:`, args);
    const result = originalMethod.apply(this, args);
    console.log(`Method ${propertyKey} returned:`, result);
    return result;
  };
}

class MathOps {
  @LogCall
  multiply(a: number, b: number): number {
    return a * b;
  }
}

const math = new MathOps();
math.multiply(4, 5);
// Console output:
// Calling multiply with arguments: [4, 5]
// Method multiply returned: 20
```

Benefits:

- Keeps logging concerns separate from business logic.
- Easily reusable on any method by adding the decorator.

15.3.2 Validating Input Data or Property Values

You can use property or parameter decorators to enforce validation rules. For example, marking a property as required and checking its value:

```
const requiredMetadataKey = Symbol("required");

function Required(target: any, propertyKey: string) {
  let requiredProps: string[] = Reflect.getMetadata(requiredMetadataKey, target) || [];
  requiredProps.push(propertyKey);
  Reflect.defineMetadata(requiredMetadataKey, requiredProps, target);
}

function validate(obj: any): boolean {
  const requiredProps: string[] = Reflect.getMetadata(requiredMetadataKey, obj) || [];
  return requiredProps.every(prop => obj[prop] !== undefined && obj[prop] !== null);
}

class User {
  @Required
  username!: string;

  @Required
  email!: string;
}

const user = new User();
user.username = "alice";

console.log("Is valid:", validate(user)); // false because email is missing
```

Note: This example uses `reflect-metadata` library, which is often required for advanced decorator metadata management.

Benefits:

- Validation rules are declared close to the data definition.
- Avoids repetitive manual validation code.

15.3.3 Attaching Metadata for Runtime Reflection

Decorators can attach metadata to classes or methods, which can then be inspected at runtime for dynamic behavior, such as dependency injection, serialization, or API documentation.

```
function Route(path: string) {
  return function (target: any, propertyKey: string) {
    Reflect.defineMetadata("route:path", path, target, propertyKey);
  };
}

class ApiController {
  @Route("/users")
  getUsers() {
    // implementation
  }
}

// Later, inspect metadata
```

```
const routePath = Reflect.getMetadata("route:path", ApiController.prototype, "getUsers");
console.log("Route path for getUsers:", routePath); // "/users"
```

15.3.4 Why Use Decorators?

- **Separation of Concerns:** Keep logging, validation, or metadata logic separate from core business logic.
- **Reusability:** Write once, apply everywhere.
- **Less Boilerplate:** Reduce repetitive code by declaring behavior declaratively.
- **Strong Typing Support:** TypeScript checks ensure decorators are applied correctly and consistently.

Decorators transform the way you write clean, maintainable, and modular TypeScript applications by allowing declarative enhancements that are both powerful and easy to reason about.

Chapter 16.

TypeScript with Third-Party Libraries

1. Using Declaration Files (`.d.ts`)
2. Working with DefinitelyTyped (`@types`)
3. TypeScript in Node.js and Express

16 TypeScript with Third-Party Libraries

16.1 Using Declaration Files (`.d.ts`)

When you use JavaScript libraries in your TypeScript project, sometimes those libraries don't come with built-in type definitions. This is where **declaration files** come in handy.

16.1.1 What Are Declaration Files?

Declaration files, with the `.d.ts` extension, describe the types and shapes of existing JavaScript code **without** providing any implementation. They act like a bridge between JavaScript libraries and TypeScript, telling the compiler what types exist, what functions you can call, and what properties objects have.

- They enable **type safety** when using JavaScript libraries.
- They provide better **code completion**, **intellisense**, and **error checking** in your editor.
- They do **not** affect runtime code; they are only used during compile time.

16.1.2 Structure of a Declaration File

A typical `.d.ts` file includes:

- **Type declarations** for functions, classes, and variables.
- **Interfaces** describing object shapes.
- **Modules** declaring the external library or namespace.

Example:

```
// myLibrary.d.ts

declare module "myLibrary" {
  export function greet(name: string): string;
  export interface User {
    id: number;
    name: string;
  }
}
```

This file tells TypeScript that the `"myLibrary"` module exports a function `greet` and an interface `User`.

16.1.3 How to Use Declaration Files

If you have a `.d.ts` file for a library you want to use, you can:

1. Place the `.d.ts` file somewhere TypeScript can find it (e.g., in a `types/` folder).
2. Reference it in your `tsconfig.json` via the `typeRoots` or `include` settings if needed.
3. Import and use the library as usual, with full type support.

Example:

```
import { greet, User } from "myLibrary";

const user: User = { id: 1, name: "Alice" };
console.log(greet(user.name)); // Type-safe usage!
```

16.1.4 Benefits

- **Improved Editor Experience:** Autocomplete and inline documentation appear for the library's functions and types.
- **Compile-Time Safety:** Errors are caught early if you misuse the library API.
- **Seamless Integration:** Use existing JavaScript libraries without rewriting or losing TypeScript benefits.

Summary

Declaration files are essential for working smoothly with JavaScript libraries that don't provide native TypeScript support. They give you all the advantages of static typing and tooling without altering your runtime code.

Ready to move on to working with DefinitelyTyped packages or integrating TypeScript with Node.js and Express?

16.2 Working with DefinitelyTyped (`@types`)

When working with popular JavaScript libraries in TypeScript, you often need type definitions to enable type safety and better tooling. This is where **DefinitelyTyped** comes in.

16.2.1 What is DefinitelyTyped?

DefinitelyTyped is a large, community-driven repository of TypeScript type definitions for many popular JavaScript libraries that don't include their own types. These type definitions are published as separate npm packages under the `@types` namespace.

16.2.2 Installing Type Definitions with `@types`

You can install type definitions using npm or yarn. For example, if you need to use the popular utility library **Lodash** with full type safety, you would install its types like this:

```
npm install lodash @types/lodash
```

or with yarn:

```
yarn add lodash @types/lodash
```

16.2.3 How `@types` Enhances Development

By installing `@types` packages, TypeScript:

- **Infers correct types** for library functions and objects.
- Provides **IntelliSense and autocompletion** in your code editor.
- Catches **compile-time errors** if you misuse the library API.
- Enables **refactoring and navigation** features with type knowledge.

Example with Lodash:

```
import _ from "lodash";

const nums = [1, 2, 3, 4];
const doubled = _.map(nums, n => n * 2);  // TypeScript knows `doubled` is number[]
console.log(doubled);
```

Because the types are installed, you get full type safety and editor support for Lodash's API.

16.2.4 Common Usage Tips and Troubleshooting

- **Always install the matching version:** Check the version compatibility between the library and its `@types` package.
- **No need to import types separately:** When you import the library, TypeScript automatically uses the installed types.
- **Sometimes types are built-in:** Some libraries include their own type declarations, so `@types` packages are not required.
- **If types are missing:** Search for the library in the DefinitelyTyped repository or on npm under `@types`.
- **Handling conflicts:** If multiple types conflict, you may need to update or clean your `node_modules`.

16.2.5 Summary

Using `@types` from DefinitelyTyped is the easiest way to add type support for popular JavaScript libraries. It makes your TypeScript code safer, easier to write, and better supported by your editor.

16.3 TypeScript in Node.js and Express

Using TypeScript with Node.js and Express lets you build scalable, maintainable server-side applications with type safety. In this section, we'll walk through setting up a minimal Express server using TypeScript and demonstrate common patterns with full type annotations.

16.3.1 Setting Up a TypeScript Express Project

1. **Initialize your project and install dependencies:**

```
npm init -y

npm install express
npm install --save-dev typescript @types/node @types/express ts-node nodemon
```

- `typescript`: The TypeScript compiler
- `@types/node` and `@types/express`: Type definitions for Node.js and Express
- `ts-node`: Runs TypeScript files directly without manual compilation
- `nodemon`: Automatically restarts the server on file changes

2. **Create a `tsconfig.json`:**

Run `npx tsc --init` to generate a default config, then adjust these important options for Node.js:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true
  }
}
```

16.3.2 Writing a Minimal Express Server in TypeScript

Create a `src/index.ts` file:

```
import express, { Request, Response, NextFunction } from 'express';

const app = express();
const PORT = 3000;

// Middleware example: simple logger
app.use((req: Request, res: Response, next: NextFunction) => {
  console.log(`${req.method} ${req.path}`);
  next();
});

// Basic route with typed req and res
app.get('/', (req: Request, res: Response) => {
  res.send('Hello from TypeScript + Express!');
});

// Route with query parameters
app.get('/greet', (req: Request, res: Response) => {
  const name = req.query.name as string || 'Guest';
  res.send(`Hello, ${name}!`);
});

app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

16.3.3 Running Your TypeScript Express Server

Add scripts to your `package.json` for convenience:

```
"scripts": {
  "start": "node dist/index.js",
  "build": "tsc",
  "dev": "nodemon --watch 'src/**/*.ts' --exec 'ts-node' src/index.ts"
}
```

- `npm run dev` starts the server in development mode with automatic reloads.
- `npm run build` compiles TypeScript to JavaScript in the `dist` folder.
- `npm start` runs the compiled server.

16.3.4 Key Type Annotations Explained

- `Request`, `Response`, and `NextFunction` come from Express's types.
- `req.query.name as string`: TypeScript doesn't know query types by default, so casting is common here.

- Middleware functions have three parameters: request, response, and next function for chaining.

16.3.5 Additional Express Patterns with Types

POST endpoint with typed request body

To parse JSON, add middleware and type the request body:

```
app.use(express.json());

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

app.post('/users', (req: Request<{}, {}, User>, res: Response) => {
  const user = req.body;
  res.json({ message: `User ${user.name} aged ${user.age} received.` });
});
```

Here `Request<Params, ResBody, ReqBody>` is used to specify types for route parameters, response body, and request body.

16.3.6 Summary

By combining TypeScript with Express, you gain:

- **Type safety** on requests, responses, and middleware
- Better **editor tooling** and autocomplete
- Fewer runtime bugs with strict type checking
- Cleaner, maintainable server code

With this setup, you're ready to build robust Node.js servers fully typed with TypeScript!

Chapter 17.

Configuration and Compilation

1. `tsconfig.json` Options Explained
2. Targeting ECMAScript Versions
3. Strict Mode and Recommended Flags
4. Source Maps and Debugging

17 Configuration and Compilation

17.1 `tsconfig.json` Options Explained

The `tsconfig.json` file is the central configuration file for the TypeScript compiler (`tsc`). It tells the compiler which files to include, how to process them, and what JavaScript version to output. This file helps you customize the compilation process to fit your project needs, enabling consistency and repeatability.

17.1.1 Role of `tsconfig.json`

- Defines **project root** and source files to compile.
- Specifies **compiler options** such as target JavaScript version, module system, and strictness.
- Controls **inclusion/exclusion** of files or folders.
- Improves developer experience with tooling support.

17.1.2 Key Properties in `tsconfig.json`

`compilerOptions`

This is the main section where you define how TypeScript should behave.

Common `compilerOptions` include:

Option	Description
`target`	JavaScript version to compile to (`ES5`, `ES2015`, etc.)
`module`	Module system used (`commonjs`, `esnext`, `amd`, etc.)
`outDir`	Output directory for compiled JavaScript
`rootDir`	Root directory of input source files
`strict`	Enables all strict type-checking options
`esModuleInterop`	Enables better interoperability for default imports
`sourceMap`	Generates source maps for debugging
`skipLibCheck`	Skips type checking of declaration files for speed

`include`

An array of glob patterns specifying which files/directories to compile.

Example:

```
"include": ["src/**/*"]
```

This means all files inside the `src` folder and subfolders are compiled.

`exclude`

An array of glob patterns specifying files/directories to exclude.

Example:

```
"exclude": ["node_modules", "dist"]
```

Typically, `node_modules` is excluded to avoid compiling dependencies, and `dist` to avoid compiling output files.

`files`

An explicit list of files to compile. Use this only when you want to compile specific files instead of entire directories.

Example:

```
"files": ["src/index.ts", "src/app.ts"]
```

17.1.3 Example: Minimal `tsconfig.json`

```
{
  "compilerOptions": {
    "target": "ES2015",
    "module": "commonjs"
  },
  "include": ["src"]
}
```

- Targets ES2015 JavaScript.
- Uses CommonJS modules (typical for Node.js).
- Compiles all files in `src`.

17.1.4 Example: More Advanced `tsconfig.json`

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "strict": true,
    "esModuleInterop": true,
    "outDir": "dist",
    "rootDir": "src",
```

```
    "sourceMap": true,
    "skipLibCheck": true
  },
  "include": ["src"],
  "exclude": ["node_modules", "dist"]
}
```

- Targets modern JavaScript (ES2020).
- Uses ESNext modules (e.g., for bundlers like Webpack).
- Enables strict type-checking for safer code.
- Supports interop between CommonJS and ES modules.
- Outputs compiled files into `dist` folder.
- Generates source maps for debugging.
- Skips checking types of external declaration files.
- Includes all files under `src`.
- Excludes dependencies and build output.

17.1.5 Summary

The `tsconfig.json` file is essential for managing your TypeScript project. By configuring its options properly, you control the compiler behavior, output format, type safety level, and which files are part of the build. Whether you start with a minimal config or an advanced setup, understanding these options helps you tailor TypeScript to your project's needs.

17.2 Targeting ECMAScript Versions

When you compile TypeScript code, the `target` option in your `tsconfig.json` determines which version of JavaScript (ECMAScript) your code will be transpiled into. This setting influences which language features are preserved as-is and which are transformed into older syntax for compatibility.

17.2.1 What Does the `target` Option Do?

- **Controls output syntax:** It tells the compiler to generate JavaScript compatible with a particular ECMAScript version.
- **Affects language feature support:** Newer JS features like arrow functions, `let`/`const`, classes, async/await, and modules are either preserved or down-leveled.
- **Impacts browser or runtime compatibility:** Older environments (like Internet Explorer) only support ES5 or earlier, while modern browsers support ES6+.

17.2.2 Common `target` Values

Target Value	Description	Supported By
`ES3`	Very old JavaScript (rarely used)	Very old browsers
`ES5`	Widely supported legacy version	IE 9+, old browsers, Node 4+
`ES6` / `ES2015`	Modern JavaScript with classes, arrow functions, etc.	Modern browsers, Node 6+
`ES2016+`	Includes async/await, newer syntax	Latest browsers and runtimes
`ESNext`	Targets latest ECMAScript features	Cutting-edge environments

17.2.3 Example: Difference Between ES5 and ES6 Output

Consider this TypeScript code:

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

When targeting ES6 (`"target": "ES6"`):

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

- Arrow function is preserved.
- Template literals are preserved.
- `const` remains intact.

When targeting ES5 (`"target": "ES5"`):

```
var greet = function (name) {
    return "Hello, " + name + "!";
};
```

- Arrow function transpiled to regular function expression.
- Template literals converted to string concatenation.
- `const` becomes `var`.

17.2.4 Trade-offs: Compatibility vs Modern Features

- **Targeting ES5:**
  - Produces code that works in older browsers and environments.
  - Requires transpilation of modern syntax, potentially larger and less performant output.
  - Needed if supporting legacy platforms.
- **Targeting ES6+ (ES2015+):**
  - Generates cleaner, more readable JavaScript using modern features.
  - Smaller output size, better performance in modern runtimes.
  - Requires runtime or build tools (like Babel or polyfills) if targeting older environments.

17.2.5 Choosing the Right Target

- **For web apps supporting older browsers:** Use `ES5`.
- **For Node.js or modern browser-only projects:** Target `ES2015` or later.
- **For libraries:** Consider your user base; maybe provide multiple builds.
- **For latest features:** Use `ESNext` but be aware of runtime compatibility.

17.2.6 Summary

The `target` compiler option is critical in balancing code modernity and compatibility. Understanding the output differences helps you configure TypeScript to produce JavaScript that runs smoothly in your intended environments while leveraging new language features when possible.

17.3 Strict Mode and Recommended Flags

TypeScript's **strict type-checking options** are a powerful feature designed to help you catch potential errors early in development and write more reliable, maintainable code. These options enforce stricter rules on how types are inferred and checked, reducing the chances of common bugs.

17.3.1 What is Strict Mode?

Enabling `"strict": true` in your `tsconfig.json` activates a group of strict type-checking options all at once. This includes settings like:

- `noImplicitAny`
- `strictNullChecks`
- `strictFunctionTypes`
- `strictBindCallApply`
- `strictPropertyInitialization`
- `noImplicitThis`
- `alwaysStrict`

You can also enable or disable these options individually if you need more granular control.

17.3.2 Key Strictness Flags Explained

Flag	Description
`noImplicitAny`	Errors on variables or parameters implicitly typed as `any`. Forces explicit typing.
`strictNullChecks`	Distinguishes `null` and `undefined` from other types, preventing accidental usage.
`strictFunctionTypes`	Enforces more precise function type checking, preventing unsafe assignments.
`strictPropertyInitialization`	Ensures class properties are initialized in the constructor or marked optional.
`noImplicitThis`	Errors if `this` is implicitly of type `any`.
`alwaysStrict`	Parses files in strict mode and emits `'use strict'`.

17.3.3 Why Enable Strict Mode?

- **Catch bugs early:** For example, if you forget to handle `null` or `undefined`, or if a variable's type is not properly declared, TypeScript will warn you.
- **Improve code clarity:** You are forced to explicitly specify types or handle edge cases, making your code self-documenting.
- **Better tooling:** Editors and IDEs can provide smarter autocomplete and error detection.
- **Prepare for scalability:** Strict typing makes your codebase safer and easier to refactor as it grows.

17.3.4 Example: Bug Caught by Strict Null Checks

```
function greet(name: string | null) {
  // Without strictNullChecks, this would allow a runtime error if name is null
  console.log("Hello, " + name.toUpperCase());
}
greet(null); // Runtime error: Cannot read property 'toUpperCase' of null
```

With `"strictNullChecks": true`, TypeScript will raise an error:

`Object is possibly 'null'.`

You must then handle the null case:

```
function greet(name: string | null) {
  if (name) {
    console.log("Hello, " + name.toUpperCase());
  } else {
    console.log("Hello, Guest!");
  }
}
```

17.3.5 Recommended Flags for Beginners Moving to Intermediate

```
{
  "compilerOptions": {
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "strictPropertyInitialization": true,
    "noImplicitThis": true,
    "alwaysStrict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noFallthroughCasesInSwitch": true
  }
}
```

These settings promote best practices such as:

- Avoiding silent type assumptions
- Handling nulls explicitly
- Enforcing correct function typings
- Keeping your code clean of unused variables or parameters
- Preventing fall-through bugs in `switch` statements

17.3.6 Summary

Enabling strict type-checking flags is a crucial step toward writing robust TypeScript code. While it might require more upfront effort, the benefits in early error detection, improved readability, and maintainability are well worth it. As you grow from a beginner to an intermediate TypeScript developer, embracing strict mode will significantly boost your confidence and productivity.

17.4 Source Maps and Debugging

When working with TypeScript, your code is compiled (or transpiled) into JavaScript before it runs. This can make debugging tricky, since the JavaScript running in the browser or Node.js is different from your original TypeScript source. **Source maps** solve this problem by providing a way to map the compiled JavaScript code back to your original TypeScript files.

17.4.1 What Are Source Maps?

A **source map** is a file that acts like a translator between the generated JavaScript and your original TypeScript code. It allows your debugging tools (such as browser dev tools or VSCode) to display TypeScript source files, show accurate line numbers, and let you set breakpoints directly in your TypeScript code—not the compiled JavaScript.

17.4.2 Enabling Source Maps in `tsconfig.json`

To generate source maps during compilation, you need to enable the `sourceMap` option in your `tsconfig.json`:

```
{
  "compilerOptions": {
    "sourceMap": true,
    "outDir": "./dist"
  },
  "include": ["src"]
}
```

- `sourceMap: true` tells TypeScript to create `.js.map` files alongside your compiled `.js` files.
- The `.map` files contain the mapping information between the TypeScript and JavaScript.

17.4.3 Example: Debugging a Simple TypeScript Program

Consider a simple TypeScript file `src/index.ts`:

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

console.log(greet("world"));
```

Compile this with:

```
tsc
```

Assuming your `tsconfig.json` has `sourceMap: true`, this will generate:

- `dist/index.js`
- `dist/index.js.map`

17.4.4 Debugging in VSCode

1. Open the compiled project folder in VSCode.
2. Set breakpoints in your original `.ts` file.
3. Use the built-in debugger with a configuration like this in `.vscode/launch.json`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}/dist/index.js",
      "sourceMaps": true,
      "outFiles": ["${workspaceFolder}/dist/**/*.js"]
    }
  ]
}
```

4. Start debugging — VSCode will map runtime execution back to your TypeScript source. You can step through the code, inspect variables, and see error locations referencing your `.ts` files.

17.4.5 Debugging in Browsers (e.g., Chrome)

1. Serve your compiled files (`dist/index.js` and `dist/index.js.map`) via a local server.
2. Open the developer tools and go to the **Sources** tab.
3. You'll see your original `.ts` files listed, allowing you to set breakpoints directly in

TypeScript.
4. When an error occurs, the stack trace points to your `.ts` source files, making debugging straightforward.

17.4.6 Practical Tips for Effective Debugging

- Always keep `sourceMap: true` enabled during development.
- Ensure your deployment pipeline includes `.map` files when debugging production issues, but be cautious exposing them publicly.
- Use meaningful source file paths (you can configure `sourceRoot` and `mapRoot` in `tsconfig.json` for complex setups).
- When exceptions happen, look at the stack traces referencing `.ts` files — this saves time pinpointing bugs.

17.4.7 Summary

Source maps bridge the gap between your original TypeScript code and the generated JavaScript, enabling smooth, precise debugging in editors and browsers. Enabling source maps in your TypeScript configuration and leveraging modern debugging tools enhances your development experience, making it easier to find and fix issues efficiently.

Chapter 18.

Building a Todo List in TypeScript

1. Creating Interfaces for Tasks
2. Managing State and Actions
3. Compiling and Running in the Browser

18 Building a Todo List in TypeScript

18.1 Creating Interfaces for Tasks

In a to-do list application, managing tasks consistently and safely is crucial. TypeScript interfaces allow us to define the expected shape of task objects, ensuring that every task has the right properties with the correct types. This helps prevent bugs and makes the code easier to understand and maintain.

18.1.1 Defining a Task Interface

Let's start by defining a basic `Task` interface that represents a single to-do item:

```
interface Task {
  id: number;            // Unique identifier for the task
  title: string;         // Description or title of the task
  completed: boolean;    // Whether the task is done or not
  dueDate?: Date;        // Optional due date for the task
}
```

Explanation:

- `id`: A unique numeric identifier to distinguish tasks.
- `title`: The text description of the task.
- `completed`: A boolean flag indicating if the task is finished.
- `dueDate`: An optional field (`?`) that may hold a `Date` object representing when the task is due.

18.1.2 How Interfaces Enforce Task Shape

Whenever we create or manipulate tasks, TypeScript uses this interface to check that objects conform to the expected structure.

Example: Creating a Task

```
const task1: Task = {
  id: 1,
  title: "Buy groceries",
  completed: false,
  dueDate: new Date("2025-07-01"),
};
```

This is valid because it matches the `Task` interface exactly.

Example: Missing Required Field (Error)

```
const task2: Task = {
  id: 2,
  completed: false,
};
// Error: Property 'title' is missing in type '{ id: number; completed: boolean; }'
// but required in type 'Task'.
```

Here, TypeScript immediately reports an error because `title` is missing.

18.1.3 Using Interfaces Across the Codebase

By using the `Task` interface everywhere in your app—such as in functions, state management, or UI components—you gain consistent typing and auto-completion support.

Function Parameter Example

```
function toggleTaskCompletion(task: Task): Task {
  return { ...task, completed: !task.completed };
}
```

This function accepts a `Task` object, safely toggles its completion status, and returns a new `Task`.

18.1.4 Summary

- Interfaces clearly define what properties a task must have.
- Optional properties like `dueDate` provide flexibility without sacrificing type safety.
- Using interfaces throughout your app helps catch errors early and improves code readability.

Defining interfaces upfront is the first step to building a robust, type-safe to-do list application!

Ready to manage your tasks and state next? Let's move on to the next section!

18.2 Managing State and Actions

In any to-do list application, managing the state of tasks—such as adding, updating, deleting, and toggling their completion status—is essential. TypeScript helps us keep these operations type-safe and predictable by defining clear interfaces for both the state and the actions that modify it.

18.2.1 Defining the State and Actions

State Interface

The state will be an array of `Task` objects, representing the current list of tasks:

```
interface Task {
  id: number;
  title: string;
  completed: boolean;
  dueDate?: Date;
}

type TaskState = Task[];
```

Action Types

Next, define the possible actions that can be performed on the state. We'll use a discriminated union of action objects:

```
type TaskAction =
  | { type: "ADD"; payload: Task }
  | { type: "UPDATE"; payload: Task }
  | { type: "DELETE"; payload: number }  // task id
  | { type: "TOGGLE"; payload: number }; // task id
```

Each action has a `type` describing what it does and a `payload` carrying the necessary data.

18.2.2 Updating State with Functions

Here is a function to manage the state immutably based on the action received:

```
function taskReducer(state: TaskState, action: TaskAction): TaskState {
  switch (action.type) {
    case "ADD":
      // Add a new task to the list
      return [...state, action.payload];

    case "UPDATE":
      // Update an existing task by matching id
      return state.map(task =>
        task.id === action.payload.id ? { ...task, ...action.payload } : task
      );

    case "DELETE":
      // Remove a task by id
      return state.filter(task => task.id !== action.payload);

    case "TOGGLE":
      // Toggle the completed status of a task by id
      return state.map(task =>
        task.id === action.payload ? { ...task, completed: !task.completed } : task
      );
```

```
    default:
      return state;
  }
}
```

Explanation:

- Each case creates a new array without mutating the original, preserving immutability.
- `ADD` appends the new task.
- `UPDATE` finds and merges changes into the matching task.
- `DELETE` filters out the task by `id`.
- `TOGGLE` flips the `completed` status.

18.2.3 Using the Reducer Function

```
let tasks: TaskState = [];

// Add a task
tasks = taskReducer(tasks, {
  type: "ADD",
  payload: { id: 1, title: "Learn TypeScript", completed: false },
});

// Toggle completion
tasks = taskReducer(tasks, { type: "TOGGLE", payload: 1 });

console.log(tasks);
// Output:
// [{ id: 1, title: "Learn TypeScript", completed: true }]
```

18.2.4 Benefits of This Approach

- **Type safety:** TypeScript ensures only valid actions and state shapes are used.
- **Predictable updates:** The reducer pattern keeps state updates clear and centralized.
- **Immutability:** By creating new state arrays, you avoid accidental mutations which can cause bugs.

18.2.5 Summary

- Define `TaskState` and `TaskAction` types for strong typing.
- Use a reducer function to handle state changes immutably and type-safely.
- This pattern scales well as the app grows in complexity and ensures maintainable code.

Next, we'll see how to compile this code and run the to-do app smoothly in a browser environment.

Ready to take your TypeScript to the browser? Let's continue!

18.3 Compiling and Running in the Browser

Once you have your TypeScript code ready—like the interfaces, state management, and UI logic—the next step is to compile it into JavaScript so browsers can run it.

18.3.1 Step 1: Setting Up the Project Structure

Here's a simple folder structure for your to-do app:

```
todo-app/
+-- index.html
+-- src/
    +-- app.ts
+-- dist/
    +-- app.js
+-- tsconfig.json
```

- `src/app.ts`: Your TypeScript source code.
- `dist/app.js`: The compiled JavaScript output.
- `index.html`: The HTML file loading the JavaScript.
- `tsconfig.json`: TypeScript configuration.

18.3.2 Step 2: Creating `tsconfig.json`

Create a `tsconfig.json` file to configure compilation:

```
{
  "compilerOptions": {
    "target": "ES6",            // Modern JavaScript target
    "module": "ES6",            // Use ES modules
    "outDir": "./dist",         // Output directory for compiled JS
    "rootDir": "./src",         // Source root directory
    "strict": true,             // Enable strict type-checking
    "sourceMap": true           // Generate source maps for debugging
  },
  "include": ["src/**/*"]
}
```

This setup compiles all `.ts` files inside `src/` and outputs `.js` files into `dist/`.

18.3.3 Step 3: Compiling the Code

Open your terminal inside the project root and run:

```
npx tsc
```

This command compiles your TypeScript files according to `tsconfig.json`. Alternatively, to auto-compile on file changes, run:

```
npx tsc --watch
```

18.3.4 Step 4: Writing `index.html`

Create a simple HTML file that includes the compiled JavaScript:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>TypeScript Todo App</title>
</head>
<body>
  <h1>Todo List</h1>
  <div id="app"></div>

  <script type="module" src="./dist/app.js"></script>
</body>
</html>
```

- The `type="module"` attribute allows modern browsers to support ES6 modules.
- The script points to your compiled file in `dist/app.js`.

18.3.5 Step 5: Running the App Locally

To test your app in the browser, you need a simple web server (browsers block module scripts loaded via `file://` protocol):

- **Using VS Code:** You can install the "Live Server" extension and right-click `index.html` → "Open with Live Server".
- **Using Python:** Run `python -m http.server` in your project root and open `http://localhost:8000`.
- **Using Node.js:** You can install `serve` with `npm install -g serve` and run `serve` in your project directory.

18.3.6 Step 6: Interacting with Your Todo App

Here's a minimal example snippet inside `src/app.ts` that interacts with the DOM and uses your types:

```
interface Task {
  id: number;
  title: string;
  completed: boolean;
}

let tasks: Task[] = [];

const appDiv = document.getElementById("app") as HTMLDivElement;

function renderTasks() {
  appDiv.innerHTML = tasks
    .map(task => `<div>
      <input type="checkbox" ${task.completed ? "checked" : ""} />
      ${task.title}
    </div>`)
    .join("");
}

// Add a sample task and render
tasks.push({ id: 1, title: "Learn TypeScript", completed: false });
renderTasks();
```

When you open your app in the browser, you should see the task rendered with a checkbox.

18.3.7 Why This Matters

- **Type Safety in UI:** Your tasks and interactions remain strongly typed.
- **Source Maps:** Debug TypeScript code directly in the browser dev tools if `sourceMap` is enabled.
- **Modular Code:** Using modules and modern syntax keeps your code clean and maintainable.

18.3.8 Optional: Using Bundlers

For larger projects, you may want to use bundlers like Webpack, Rollup, or Vite to:

- Bundle multiple modules into one script.
- Support CSS, images, and other assets.
- Optimize for production.

However, for a beginner's to-do list app, the simple setup above is often enough.

18.4 Summary

- Organize source files under `src/` and compile to `dist/`.
- Configure `tsconfig.json` to target ES6 and enable source maps.
- Use `tsc` or `tsc --watch` to compile your code.
- Serve your files via a local web server to avoid CORS issues.
- Load your compiled JavaScript modules in `index.html`.
- Enjoy type-safe, maintainable TypeScript code running smoothly in the browser!

You're now ready to build more interactive features for your to-do list with confidence in your types and tooling.

Ready to dive deeper? Let's continue building!

Chapter 19.

Simple REST API with TypeScript and Node

1. Setting up with `Express`
2. Routing and Type Safety
3. Working with JSON Data

19 Simple REST API with TypeScript and Node

19.1 Setting up with `Express`

In this section, you'll learn how to set up a basic REST API server using **Express** and **TypeScript**. Express is a popular Node.js web framework that makes building web servers and APIs straightforward.

19.1.1 Step 1: Initialize the Project

First, create a new folder for your project and initialize a Node.js project with:

```
mkdir ts-express-api
cd ts-express-api
npm init -y
```

This creates a `package.json` with default settings.

19.1.2 Step 2: Install Dependencies

Next, install the required dependencies:

```
npm install express
npm install --save-dev typescript ts-node @types/node @types/express
```

- `express`: The core Express library.
- `typescript`: The TypeScript compiler.
- `ts-node`: Allows running TypeScript files directly without manual compilation.
- `@types/express` and `@types/node`: Type definitions for Express and Node.js, enabling type safety and IntelliSense.

19.1.3 Step 3: Configure TypeScript

Create a `tsconfig.json` file in your project root with the following minimal configuration:

```
{
  "compilerOptions": {
    "target": "ES6",                // Target modern JavaScript version
    "module": "CommonJS",           // Use CommonJS modules (Node.js default)
    "outDir": "./dist",             // Output directory for compiled JavaScript
    "rootDir": "./src",             // Root directory of source files
    "strict": true,                 // Enable strict type checking
    "esModuleInterop": true         // Enable compatibility with ES modules
  },
```

```
  "include": ["src/**/*"]
}
```

This config ensures TypeScript compiles your source files in `src/` to `dist/`.

19.1.4 Step 4: Create the Server File

Inside the `src` folder (create it if it doesn't exist), create a file `index.ts`:

```
import express, { Request, Response } from "express";

const app = express();
const PORT = 3000;

// Middleware example: JSON body parsing
app.use(express.json());

// Define a simple route that returns "Hello World"
app.get("/", (req: Request, res: Response) => {
  res.send("Hello World");
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running at http://localhost:${PORT}`);
});
```

19.1.5 Step 5: Running the Server

Option 1: Run with `ts-node` (no manual compile)

Install `ts-node` globally or use npx:

```
npx ts-node src/index.ts
```

This runs your TypeScript file directly.

Option 2: Compile and run with Node.js

Compile your code:

```
npx tsc
```

This compiles TypeScript files to JavaScript in the `dist/` folder.

Then run:

```
node dist/index.js
```

19.1.6 Whats Happening Here?

- **Express app**: You create an Express application instance with `express()`.
- **Middleware**: `app.use(express.json())` adds middleware that parses JSON bodies of incoming requests.
- **Route handler**: `app.get("/", ...)` registers a handler for HTTP GET requests on the root URL `/`.
- **Listening**: `app.listen(PORT, ...)` starts the server on the specified port and logs a confirmation.

19.1.7 Summary

You've successfully set up a TypeScript-powered Express server that responds with "Hello World". This setup is the foundation for building REST APIs with type safety, auto-completion, and modern JavaScript features.

Next, we will explore how to add typed routing and handle HTTP methods with more precision.

Try it out: Start your server and open `http://localhost:3000` in your browser to see your "Hello World" message!

19.2 Routing and Type Safety

In Express, routing refers to defining how your server responds to different HTTP requests and URLs. When using TypeScript, you can leverage its type system to ensure your route handlers handle requests and responses correctly — preventing common runtime errors by catching type issues at compile time.

19.2.1 Typing Express Route Handlers

Express route handlers receive `Request` and `Response` objects, which you can import and type explicitly for full type safety:

```
import { Request, Response } from "express";
```

By default, these objects are generic, allowing you to specify types for:

- Path parameters (`params`)
- Query parameters (`query`)
- Request body (`body`)

19.2.2 Example 1: Typed GET Route with Query Parameters

Suppose you have a route that accepts a search query via URL parameters:

```
import express, { Request, Response } from "express";

const app = express();

interface SearchQuery {
  term: string;
  page?: string;  // optional query parameter
}

app.get("/search", (req: Request<{}, {}, {}, SearchQuery>, res: Response) => {
  const searchTerm = req.query.term;
  const page = req.query.page || "1";

  res.send(`Searching for "${searchTerm}" on page ${page}`);
});
```

- `Request<Params, ResBody, ReqBody, ReqQuery>`:
  - Here, `Params` and `ReqBody` are empty `{}` since no path parameters or body are expected.
  - `ReqQuery` is typed as `SearchQuery` to access typed query params.
- This ensures that accessing `req.query.term` is type-safe.

19.2.3 Example 2: Typed POST Route with Request Body

For a POST endpoint expecting JSON data:

```
interface NewUser {
  username: string;
  email: string;
  age?: number; // optional field
}

app.use(express.json()); // Middleware to parse JSON bodies

app.post("/users", (req: Request<{}, {}, NewUser>, res: Response) => {
```

```
  const user = req.body;
  res.status(201).send(`User ${user.username} created successfully.`);
});
```

- Here, `Request<Params, ResBody, ReqBody>`:
  - `ReqBody` is typed as `NewUser`.
- TypeScript ensures you access valid properties on `req.body`.

19.2.4 Example 3: Parameterized Routes with Path Parameters

To define routes with dynamic URL segments (like user IDs):

```
interface UserParams {
  userId: string;  // URL param is always a string
}

app.get("/users/:userId", (req: Request<UserParams>, res: Response) => {
  const userId = req.params.userId;
  res.send(`User profile for user ID: ${userId}`);
});
```

- `Request<UserParams>` defines the shape of `req.params`.
- TypeScript prevents mistakes such as accessing non-existent parameters.

19.2.5 How TypeScript Helps

- **Compile-time checks**: Catch typos in parameter names or misuse of request/response properties before running your app.
- **IntelliSense support**: Editors like VSCode offer autocomplete and inline documentation, speeding development.
- **Safer code**: You avoid runtime crashes caused by undefined or incorrectly typed data.
- **Self-documenting code**: Types serve as documentation for expected request shapes.

19.2.6 Summary

Here is a quick reference of `Request` generics:

```
Request<
  Params = core.ParamsDictionary,
  ResBody = any,
  ReqBody = any,
  ReqQuery = core.Query
```

```
>
```

- `Params`: URL path parameters (e.g., `/users/:id`)
- `ResBody`: Expected response body type (rarely typed in simple APIs)
- `ReqBody`: Request body type (for POST/PUT)
- `ReqQuery`: Query string parameters

19.2.7 Final Notes

- Always enable middleware like `express.json()` to parse JSON bodies before accessing `req.body`.
- Define interfaces for request parts to make your API predictable and maintainable.
- With strong typing, your Express routes become more reliable and easier to refactor safely.

19.3 Working with JSON Data

Handling JSON data is essential in REST APIs, as JSON is the most common format for sending and receiving structured data over HTTP. In this section, we'll see how to accept JSON request bodies, validate their structure with TypeScript interfaces, and send typed JSON responses.

19.3.1 Parsing JSON Request Bodies

Express does not parse JSON bodies automatically. To enable this, use the built-in middleware:

```
import express from "express";

const app = express();

app.use(express.json());  // Enables JSON body parsing
```

The `express.json()` middleware reads the incoming request body and parses it into a JavaScript object, which becomes available on `req.body`.

19.3.2 Defining Interfaces for JSON Data

Using interfaces, you can clearly specify the expected shape of the incoming and outgoing JSON data.

For example, let's define a `Task` interface:

```
interface Task {
  id: number;
  title: string;
  completed: boolean;
  dueDate?: string;  // Optional ISO date string
}
```

19.3.3 Accepting JSON Data in Requests

Here's a POST endpoint that accepts a new task in JSON format:

```
import { Request, Response } from "express";

app.post("/tasks", (req: Request<{}, {}, Task>, res: Response<Task>) => {
  const newTask = req.body;

  // Basic validation example
  if (!newTask.title) {
    return res.status(400).json({ error: "Title is required" } as any);
  }

  // Here you would typically save the task to a database
  console.log("Received task:", newTask);

  // Respond with the created task and 201 status code
  res.status(201).json(newTask);
});
```

- The `Request` type's third generic parameter `<Task>` ensures `req.body` matches the `Task` interface.
- The response type `Response<Task>` indicates that JSON returned will conform to `Task`.
- TypeScript helps you catch missing or mismatched properties during development.

19.3.4 Sending Typed JSON Responses

When responding with JSON data, TypeScript allows you to annotate the response type:

```
app.get("/tasks/:id", (req: Request<{ id: string }>, res: Response<Task | { error: string }>) => {
  const id = Number(req.params.id);

  // Simulate fetching a task
```

```
  const task: Task | undefined = id === 1 ? {
    id: 1,
    title: "Learn TypeScript",
    completed: false,
  } : undefined;

  if (!task) {
    return res.status(404).json({ error: "Task not found" });
  }

  res.json(task);
});
```

This ensures the response is either a `Task` object or an error message, helping clients and developers understand the possible responses.

19.3.5 Basic Validation Tips

While TypeScript enforces types at compile time, it does **not** validate data at runtime. To ensure data integrity:

- Use simple runtime checks (like in the example above).
- Consider libraries like `zod` or `io-ts` for more comprehensive runtime validation.
- Always validate external inputs before processing or saving.

19.3.6 Summary

- Use `express.json()` middleware to parse JSON bodies.
- Define interfaces to type request and response JSON data.
- Use TypeScript generics on `Request` and `Response` to enforce type safety.
- Perform basic runtime validation to prevent invalid data processing.
- Typed JSON handling improves reliability and developer confidence.

Chapter 20.

TypeScript in the Browser

1. DOM Manipulation and Events
2. Fetch API with Strong Typing
3. Form Handling and Validation

20 TypeScript in the Browser

20.1 DOM Manipulation and Events

Manipulating the DOM and responding to user interactions are fundamental tasks in web development. TypeScript enhances these tasks by providing strong typing and helpful tooling when working with DOM APIs.

20.1.1 Selecting DOM Elements with Type Assertions

Methods like `document.getElementById` and `document.querySelector` return general types such as `HTMLElement | null` or `Element | null`. TypeScript requires you to handle the possibility of `null` and sometimes narrow down the element type to access specific properties safely.

Example: Selecting and Manipulating an Input Element

```
// Select the input element by ID
const input = document.getElementById("username") as HTMLInputElement | null;

if (input) {
  // Access .value safely, because input is asserted as HTMLInputElement
  input.value = "Hello, TypeScript!";
} else {
  console.error("Input element not found.");
}
```

Here we use a **type assertion** (`as HTMLInputElement`) to tell TypeScript the exact type of the element, enabling access to `.value` property which is specific to inputs.

Example: Using `querySelector` to Select a Button

```
const button = document.querySelector(".submit-btn") as HTMLButtonElement | null;

if (button) {
  button.textContent = "Submit Form";
}
```

20.1.2 Adding Event Listeners with Typed Handlers

When adding event listeners, TypeScript allows you to specify the type of the event parameter to get accurate autocomplete and type safety.

Example: Handling a Click Event

```
const button = document.getElementById("clickMe") as HTMLButtonElement | null;

if (button) {
  button.addEventListener("click", (event: MouseEvent) => {
    console.log("Button clicked!", event);
    // Change button text on click
    button.textContent = "Clicked!";
  });
}
```

Example: Handling Keyboard Events on an Input

```
const input = document.getElementById("username") as HTMLInputElement | null;

if (input) {
  input.addEventListener("keydown", (event: KeyboardEvent) => {
    if (event.key === "Enter") {
      console.log("Enter key pressed!", input.value);
    }
  });
}
```

20.1.3 Practical Demo: Live Content Update

Here's a quick demo that changes the content of a `<div>` when a button is clicked:

```
const output = document.getElementById("output") as HTMLDivElement | null;
const button = document.getElementById("updateBtn") as HTMLButtonElement | null;

if (output && button) {
  button.addEventListener("click", () => {
    output.textContent = "Content updated at " + new Date().toLocaleTimeString();
  });
}
```

20.1.4 Summary

- Use type assertions to specify exact element types (e.g., `HTMLInputElement`, `HTMLButtonElement`).
- Always check for `null` to avoid runtime errors.
- Annotate event handler parameters with appropriate event types (`MouseEvent`, `KeyboardEvent`) for safety and tooling.
- Typed DOM manipulation helps catch mistakes early and improves code clarity.

Next, we'll explore making HTTP requests with the Fetch API, combined with strong TypeScript typings for safer and clearer code.

20.2 Fetch API with Strong Typing

Fetching data from external APIs is a common task in web development. TypeScript can greatly improve safety and clarity by strongly typing the response data you expect to receive from these calls.

20.2.1 Using Fetch with Typed JSON Responses

The Fetch API returns a `Promise<Response>`, and parsing JSON data with `.json()` returns `Promise<any>`. To get strong typing, you define interfaces that describe the expected structure of the JSON data.

Step 1: Define the Response Interface

Suppose we are fetching user data from a mock API endpoint:

```
interface User {
  id: number;
  name: string;
  username: string;
  email: string;
}
```

Step 2: Create an Async Function to Fetch and Parse Data

```
async function fetchUsers(): Promise<User[]> {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/users');

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    // Parse JSON with explicit typing
    const data: User[] = await response.json();

    return data;
  } catch (error) {
    console.error('Failed to fetch users:', error);
    return []; // Return an empty array or handle appropriately
  }
}
```

Step 3: Use the Typed Data Safely

```
fetchUsers().then(users => {
  users.forEach(user => {
    console.log(`${user.name} (${user.email})`);
  });
});
```

20.2.2 Benefits of Strong Typing with Fetch

- **Type Safety**: TypeScript ensures the properties you access on the data exist and have correct types.
- **Editor Support**: Autocomplete and inline documentation make working with API data easier.
- **Early Error Detection**: Mistyped property names or incorrect assumptions about the API response are caught at compile time.
- **Clearer Code**: Explicit interfaces document the shape of the data for yourself and others.

20.2.3 Handling Errors Gracefully

When working with real-world APIs, network errors or unexpected responses can happen. Use `try/catch` to handle these cases and type guards to protect against unexpected data shapes.

```
async function fetchUserById(id: number): Promise<User | null> {
  try {
    const response = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

    if (!response.ok) {
      throw new Error(`User not found, status: ${response.status}`);
    }

    const user: User = await response.json();
    return user;
  } catch (error) {
    console.error('Error fetching user:', error);
    return null;
  }
}
```

20.2.4 Summary

- Define interfaces for the expected JSON data shape.
- Use `fetch()` inside async functions and type the parsed JSON.
- Handle HTTP and network errors properly.
- Typed responses reduce bugs and improve developer experience.

Next, we'll learn how to handle form inputs and validate user data with TypeScript.

20.3 Form Handling and Validation

Building interactive forms is a key part of web development. With TypeScript, you can handle form data and validations more safely and clearly by leveraging strong typing and modern DOM APIs.

20.3.1 Creating a Simple HTML Form

Here's a basic form for adding a new task with a title and due date:

```
<form id="taskForm">
  <label>
    Task Title:
    <input type="text" id="title" name="title" required />
  </label>
  <br />
  <label>
    Due Date:
    <input type="date" id="dueDate" name="dueDate" />
  </label>
  <br />
  <button type="submit">Add Task</button>
  <p id="errorMessage" style="color: red;"></p>
</form>
```

20.3.2 Accessing Form Inputs in TypeScript

To work with this form in TypeScript, you select the form and inputs, and assert the correct element types:

```
const form = document.getElementById('taskForm') as HTMLFormElement;
const titleInput = document.getElementById('title') as HTMLInputElement;
const dueDateInput = document.getElementById('dueDate') as HTMLInputElement;
const errorMessage = document.getElementById('errorMessage') as HTMLParagraphElement;
```

20.3.3 Handling Form Submission

You can listen for the `submit` event, prevent the default behavior, and safely retrieve and validate input values:

```
form.addEventListener('submit', (event) => {
  event.preventDefault(); // Prevent page reload

  const title = titleInput.value.trim();
  const dueDate = dueDateInput.value; // empty string if no date selected
```

```
  // Built-in HTML validation already checks required fields,
  // but additional custom validation can improve UX
  if (title.length === 0) {
    errorMessage.textContent = 'Task title is required.';
    return;
  }

  if (dueDate && isNaN(Date.parse(dueDate))) {
    errorMessage.textContent = 'Invalid due date.';
    return;
  }

  errorMessage.textContent = ''; // Clear previous errors

  // Use validated values (example: add to task list)
  console.log('Adding task:', { title, dueDate });

  // Reset form after submission
  form.reset();
});
```

20.3.4 Built-in vs Custom Validation

Built-in Validation

- HTML attributes like `required`, `min`, `max`, `pattern` enforce basic checks automatically.
- Browsers provide instant feedback and block submission if invalid.

Custom Validation with TypeScript

- Allows complex rules like conditional logic or cross-field checks.
- Gives full control over error messages and UI updates.
- Enables type-safe access to inputs and consistent validation logic.

20.3.5 Improving User Experience with Feedback

Update the UI dynamically based on validation:

```
titleInput.addEventListener('input', () => {
  if (titleInput.validity.valid) {
    errorMessage.textContent = '';
  }
});
```

You can expand this pattern to provide inline feedback, disable buttons when invalid, or highlight inputs.

20.3.6 Summary

- Use type assertions like `as HTMLInputElement` to work safely with form elements.
- Combine built-in HTML validation with TypeScript custom checks.
- Handle form submissions by preventing default and validating inputs.
- Provide clear, type-safe feedback to users for better UX.

With these techniques, your forms become robust, user-friendly, and maintainable — all while leveraging the power of TypeScript's type system.

Chapter 21.

TypeScript and React (Intro)

1. Typing Props and State
2. Functional Components and Hooks
3. Component Reusability with Generics

21 TypeScript and React (Intro)

21.1 Typing Props and State

When working with React and TypeScript, properly typing your component's props and state is essential for catching errors early and improving your development experience. TypeScript helps ensure that components receive the correct data and maintain consistent internal state.

21.1.1 Typing Props in Functional Components

You can define an interface or type describing the props your component expects. This gives you strong guarantees about the shape and types of the data passed into components.

```
import React from 'react';

interface GreetingProps {
  name: string;
  age?: number; // optional prop
}

const Greeting: React.FC<GreetingProps> = ({ name, age }) => {
  return (
    <div>
      <p>Hello, {name}!</p>
      {age && <p>You are {age} years old.</p>}
    </div>
  );
};
```

Notes:

- `name` is required.
- `age` is optional (`?`).
- Using `React.FC<Props>` adds type checking for props and also includes the `children` prop by default.

21.1.2 Default Props with TypeScript

To provide default values for optional props, you can use default parameters or default values:

```
const Greeting: React.FC<GreetingProps> = ({ name, age = 18 }) => {
  return (
    <div>
      <p>Hello, {name}!</p>
      <p>You are {age} years old.</p>
    </div>
  );
};
```

Here, if `age` is not passed, it defaults to 18.

21.1.3 Typing State with `useState`

When using React hooks like `useState`, you can specify the type of your state variable for full type safety:

```
import React, { useState } from 'react';

const Counter: React.FC = () => {
  // State is typed as number
  const [count, setCount] = useState<number>(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
};
```

TypeScript infers the type from the initial value (here `0`), but explicitly typing helps when initial state is `null` or `undefined`.

21.1.4 Using Complex State Types

You can also use interfaces or types for more complex state objects:

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

const UserProfile: React.FC = () => {
  const [user, setUser] = useState<User | null>(null);

  // Imagine fetching user data and updating state

  return <div>{user ? user.name : 'No user logged in'}</div>;
};
```

Here, state may be `User` or `null`, and TypeScript will enforce proper handling.

21.1.5 Benefits of Typing Props and State

- **Early error detection:** TypeScript warns you if you pass wrong props or access state incorrectly.
- **Better autocompletion:** Editors provide suggestions based on your typed props and state.
- **Self-documenting code:** Interfaces clearly describe the expected data shape.
- **Improved maintainability:** Refactoring becomes safer with static types checking component contracts.

Typing props and state is a fundamental step in building robust React apps with TypeScript. It brings clarity, safety, and confidence to your component development.

21.2 Functional Components and Hooks

Functional components are the modern, recommended way to build React components. Combined with React hooks, they provide powerful features for managing state and side effects while keeping your components concise and readable.

When using TypeScript, adding type annotations to hooks and event handlers helps prevent bugs and improves your development workflow.

21.2.1 Basic Functional Component with TypeScript

A functional component is a JavaScript (or TypeScript) function that returns JSX. Using TypeScript, you define the props type as shown previously:

```
import React from 'react';

interface WelcomeProps {
  name: string;
}

const Welcome: React.FC<WelcomeProps> = ({ name }) => {
  return <h1>Welcome, {name}!</h1>;
};
```

21.2.2 Using `useState` with TypeScript

The `useState` hook lets you add state to functional components. TypeScript can infer the state type from the initial value, or you can explicitly declare it:

```
import React, { useState } from 'react';

const Counter: React.FC = () => {
  // Type inferred as number
  const [count, setCount] = useState(0);

  // Explicitly typed array of strings
  const [names, setNames] = useState<string[]>([]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>

      <button onClick={() => setNames([...names, 'New Name'])}>Add Name</button>
      <ul>{names.map((name, i) => <li key={i}>{name}</li>)}</ul>
    </div>
  );
};
```

21.2.3 Using `useEffect` with TypeScript

`useEffect` runs side effects such as data fetching or subscriptions. You can add cleanup logic by returning a function from the effect.

```
import React, { useState, useEffect } from 'react';

const Timer: React.FC = () => {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    // Cleanup function to clear the interval on unmount
    return () => clearInterval(intervalId);
  }, []); // Empty dependency array: runs once on mount

  return <p>Seconds elapsed: {seconds}</p>;
};
```

TypeScript infers types inside `useEffect`, but be sure to type any parameters or returned cleanup functions if more complex.

21.2.4 Typing Event Handlers

Event handlers are common in React. Typing them correctly avoids runtime errors and enables IntelliSense:

```
const ClickButton: React.FC = () => {
  // Mouse event on button
  const handleClick = (event: React.MouseEvent<HTMLButtonElement>) => {
    console.log('Button clicked!', event.clientX, event.clientY);
  };

  // Keyboard event on input
  const handleKeyPress = (event: React.KeyboardEvent<HTMLInputElement>) => {
    if (event.key === 'Enter') {
      alert('Enter pressed!');
    }
  };

  return (
    <>
      <button onClick={handleClick}>Click me</button>
      <input type="text" onKeyPress={handleKeyPress} />
    </>
  );
};
```

Tip: The generic event types like `React.MouseEvent`, `React.KeyboardEvent` allow you to specify the element type (`HTMLButtonElement`, `HTMLInputElement`, etc.) to access element-specific properties safely.

21.2.5 Summary

- Functional components are simple functions returning JSX.
- Hooks like `useState` and `useEffect` manage state and side effects.
- TypeScript can infer many types but explicit annotations improve clarity.
- Typing event handlers helps prevent common runtime errors.
- Proper typings lead to better tooling support and safer code.

21.3 Component Reusability with Generics

In React applications, creating reusable components is essential for scalability and maintainability. TypeScript generics allow you to write components that can operate on multiple data types without sacrificing type safety.

Generics make it possible to build flexible components that work across a variety of use cases—like rendering lists, dropdowns, or form elements—while still benefiting from autocomplete, type checking, and clear contracts.

21.3.1 Why Use Generics in React?

A generic component can accept a type parameter (e.g., `T`) that represents the shape of data it will work with. This helps:

- Enforce **type correctness** without duplicating code for different data types.
- Improve **developer experience** through IntelliSense and static analysis.
- Enable **flexible APIs** that scale across different parts of your app.

21.3.2 Example: Generic List Component

Here's a reusable `List` component that displays any type of item, given a rendering function.

```
import React from 'react';

interface ListProps<T> {
  items: T[];
  renderItem: (item: T) => React.ReactNode;
}

function List<T>({ items, renderItem }: ListProps<T>): JSX.Element {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{renderItem(item)}</li>
      ))}
    </ul>
  );
}
```

Usage with Strings:

```
const FruitList = () => {
  const fruits = ['Apple', 'Banana', 'Cherry'];

  return (
    <List
      items={fruits}
      renderItem={(fruit) => <strong>{fruit}</strong>}
    />
  );
};
```

Usage with Objects:

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

const UserList = () => {
  const users: User[] = [
```

```
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
  ];

  return (
    <List
      items={users}
      renderItem={(user) => <span>{user.name}</span>}
    />
  );
};
```

The same component now supports both `string[]` and `User[]` with full type safety and no duplication.

21.3.3 Example: Generic Dropdown Component

Let's build a dropdown that works for any list of items with a label.

```
interface DropdownProps<T> {
  options: T[];
  getLabel: (option: T) => string;
  onSelect: (option: T) => void;
}

function Dropdown<T>({ options, getLabel, onSelect }: DropdownProps<T>) {
  return (
    <select onChange={(e) => onSelect(options[e.target.selectedIndex])}>
      {options.map((option, i) => (
        <option key={i} value={i}>
          {getLabel(option)}
        </option>
      ))}
    </select>
  );
}
```

Usage:

```
interface Country {
  code: string;
  name: string;
}

const CountrySelector = () => {
  const countries: Country[] = [
    { code: 'US', name: 'United States' },
    { code: 'CA', name: 'Canada' },
  ];

  return (
    <Dropdown
      options={countries}
```

```
      getLabel={(c) => c.name}
      onSelect={(selected) => alert(`Selected: ${selected.code}`)}
    />
  );
};
```

21.3.4 Summary

- **Generic components** accept type parameters and allow maximum flexibility with strong typing.
- They work well with **lists, dropdowns, and form components**.
- TypeScript ensures that type mismatches are caught at compile time.
- This approach helps **scale applications** by building consistent, reusable building blocks.

Up next: we'll transition into more advanced React + TypeScript topics such as forms, contexts, and custom hooks.

Chapter 22.

Testing TypeScript Code

1. Unit Testing with `Jest`
2. Typing Mocks and Test Data
3. Testing Async Functions

22 Testing TypeScript Code

22.1 Unit Testing with `Jest`

Unit testing is the practice of writing code to verify that individual parts (or "units") of your application behave as expected. A unit could be a function, a class, or a module. The main goals of unit testing are to:

- Detect bugs early in development
- Ensure your code is working correctly after changes (regression testing)
- Improve the reliability and maintainability of your codebase

One of the most popular testing frameworks for JavaScript and TypeScript is **Jest**. Created by Facebook, Jest is simple to configure, fast to run, and supports TypeScript through `ts-jest`.

22.1.1 Why Use Jest?

- Built-in test runner and assertion library
- Zero-config experience for JavaScript
- TypeScript support via `ts-jest`
- Snapshot testing support
- Built-in mocking utilities

22.1.2 Setting Up Jest in a TypeScript Project

Before writing tests, you'll need to install and configure Jest to work with TypeScript.

Step 1: Install Dependencies

Open your terminal and install the following packages:

```
npm install --save-dev jest ts-jest @types/jest typescript
```

- `jest`: The core testing framework
- `ts-jest`: A TypeScript preprocessor for Jest
- `@types/jest`: Type definitions so TypeScript understands Jest functions
- `typescript`: Ensures you can compile your code

Step 2: Initialize Jest Configuration

Use the `ts-jest` CLI to configure Jest for TypeScript:

```
npx ts-jest config:init
```

This generates a basic `jest.config.js` file like the following:

```
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
};
```

Step 3: Update `tsconfig.json`

Make sure your `tsconfig.json` includes the following options:

```
{
  "compilerOptions": {
    "esModuleInterop": true,
    "module": "commonjs",
    "target": "es6",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true
  },
  "include": ["src", "**/*.test.ts"]
}
```

22.1.3 Writing Your First Test

Let's create a simple function and write a unit test for it.

Example: A Math Utility

File: `src/utils/math.ts`

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

Test File: `src/utils/math.test.ts`

```
import { add } from './math';

describe('add', () => {
  it('adds two positive numbers correctly', () => {
    expect(add(2, 3)).toBe(5);
  });

  it('adds negative numbers', () => {
    expect(add(-2, -3)).toBe(-5);
  });

  it('adds positive and negative numbers', () => {
    expect(add(5, -3)).toBe(2);
  });
```

```
});
```

22.1.4 Running Your Tests

In your `package.json`, add a script to run tests:

```
"scripts": {
  "test": "jest"
}
```

Then run:

```
npm test
```

You should see output like:

```
PASS  src/utils/math.test.ts
  add
    YES adds two positive numbers correctly
    YES adds negative numbers
    YES adds positive and negative numbers

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
```

22.1.5 Understanding Test Functions

- `describe(name, fn)`: Groups related tests together
- `it(name, fn)` or `test(name, fn)`: Defines a single test case
- `expect(actual).toBe(expected)`: Checks that the result matches the expectation

Jest also provides many other matchers, such as:

```
expect(value).toEqual(expected);  // for object comparison
expect(value).toBeTruthy();       // checks for truthy values
expect(array).toContain(item);    // checks for array membership
```

22.1.6 Why Unit Testing Matters

Writing unit tests is a key part of professional development. It may feel like extra work at first, but the benefits are significant:

- **Catches bugs early**: Tests alert you when your code breaks.

- **Refactor with confidence**: Modify existing code without fear.
- **Improves design**: Writing testable code often results in cleaner architecture.
- **Documents behavior**: Tests show how your functions are expected to behave.

22.1.7 Summary

In this section, you learned:

- What unit testing is and why it matters
- How to install and configure Jest with TypeScript using `ts-jest`
- How to write and run simple test cases
- How to use Jest's basic assertion functions

With your testing setup in place, you're ready to write robust and maintainable TypeScript code with confidence. Next, we'll explore how to handle **mocks and typing test data**.

22.2 Typing Mocks and Test Data

When writing tests, especially for larger or more complex applications, it's common to replace real functions, modules, or services with **mock** versions. These mocks simulate behavior without requiring full implementation—allowing you to focus on isolated logic. In TypeScript, adding types to mocks and test data ensures that your tests are **type-safe**, **readable**, and **maintainable**.

In this section, we'll explore:

- The differences between mocks, stubs, and test data
- How to type mock functions
- How to define reusable, typed test fixtures
- Best practices for keeping tests clear and consistent

22.2.1 Mocks vs. Stubs vs. Test Data

Term	Description
Mock	A fake function that tracks how it's used (e.g., called with which arguments)
Stub	A mock with a predefined output
Test Data	Sample input or expected output used to run assertions

22.2.2 Typed Mock Functions

In Jest, you can create mock functions using `jest.fn()`. To keep your mocks type-safe, you should specify the function signature.

Example: Typed Mock Function

Let's say you have a service:

```
// File: src/services/emailService.ts
export interface EmailService {
  sendEmail(to: string, message: string): Promise<boolean>;
}
```

In a test, you want to mock this `sendEmail` function:

```
import { EmailService } from '../services/emailService';

const mockEmailService: jest.Mocked<EmailService> = {
  sendEmail: jest.fn().mockResolvedValue(true),
};

test('should send email successfully', async () => {
  const result = await mockEmailService.sendEmail('user@example.com', 'Hello');
  expect(result).toBe(true);
  expect(mockEmailService.sendEmail).toHaveBeenCalledWith('user@example.com', 'Hello');
});
```

> YES **Tip**: Use `jest.Mocked<Type>` to mock all functions on an interface while preserving types.

22.2.3 Typing Individual Mocks

If you only need a mock for a single function:

```
const mockSendEmail = jest.fn<Promise<boolean>, [string, string]>();
```

This function:

- Returns a `Promise<boolean>`
- Accepts two arguments: `string` and `string`

You can use it in tests just like any regular mock:

```
mockSendEmail.mockResolvedValueOnce(true);
await expect(mockSendEmail('test@example.com', 'Test')).resolves.toBe(true);
```

22.2.4 Typed Test Fixtures

Test fixtures are reusable pieces of data used in tests. You can use TypeScript interfaces or types to define structure.

Example: User Type and Fixture

```
// File: src/models/user.ts
export interface User {
  id: number;
  name: string;
  email: string;
}
```

```
// File: test/fixtures/users.ts
import { User } from '../../src/models/user';

export const testUser: User = {
  id: 1,
  name: 'Alice',
  email: 'alice@example.com',
};

export const anotherUser: User = {
  id: 2,
  name: 'Bob',
  email: 'bob@example.com',
};
```

Now you can import these in your test files:

```
import { testUser } from '../fixtures/users';

test('should display user name', () => {
  expect(testUser.name).toBe('Alice');
});
```

This ensures your test data matches the structure of your real application, and reduces duplication.

22.2.5 Mocking External Modules

Sometimes, you need to mock external dependencies like APIs or database clients.

```
// File: src/api/userApi.ts
export async function fetchUser(id: number): Promise<User> {
  // Actual API call
}
```

To mock it in a test:

```
import * as userApi from '../../src/api/userApi';
import { User } from '../../src/models/user';
```

```
jest.mock('../../src/api/userApi');

const mockedFetchUser = userApi.fetchUser as jest.MockedFunction<typeof userApi.fetchUser>;

mockedFetchUser.mockResolvedValue({ id: 1, name: 'Test', email: 'test@example.com' });

test('should fetch user data', async () => {
  const user = await userApi.fetchUser(1);
  expect(user.name).toBe('Test');
});
```

YES **Best Practice**: Use `jest.MockedFunction<typeof fn>` when mocking a specific named function for full type safety.

22.2.6 Best Practices for Typed Mocks and Test Data

Here are some best practices to keep your test code readable and reliable:

1. **Use Interfaces for Fixtures**: Define interfaces or types for any objects passed into or returned from your functions.
2. **Keep Fixtures in a Separate File**: This keeps your test files short and focused.
3. **Name Mocks Clearly**: Use names like `mockEmailService`, `mockSendEmail`, etc., to make the test code self-explanatory.
4. **Use `.mockResolvedValue()` and `.mockReturnValue()`**: These methods make your mocks predictable and easy to reason about.
5. **Avoid `any`**: Always use strong typing to catch incorrect assumptions about data shapes early.
6. **Reset Mocks Between Tests**: Use `beforeEach(() => jest.clearAllMocks())` to prevent test bleed.

22.2.7 Summary

In this section, you learned:

- The difference between mocks, stubs, and test data
- How to create typed mocks with Jest
- How to define and use strongly typed test fixtures
- Best practices for keeping test code type-safe and clean

With typed mocks and structured test data, your tests become easier to write, safer to change, and more reflective of real-world scenarios.

22.3 Testing Async Functions

Asynchronous code is a core part of modern TypeScript applications, especially when dealing with tasks like HTTP requests, file operations, and timers. When writing tests for asynchronous logic, it's important to verify not only the result, but also that errors and edge cases are handled properly.

Jest provides built-in utilities like `async/await`, `.resolves`, and `.rejects` to help you write clean and readable tests for asynchronous functions.

In this section, you'll learn:

- How to test `async` functions with Jest
- How to use `.resolves` and `.rejects`
- How to mock asynchronous behavior
- How to handle errors and timeouts in async tests

22.3.1 Example 1: Testing an Async Function with `async/await`

Let's say you have a function that fetches a user profile:

```
// File: src/services/userService.ts
export interface User {
  id: number;
  name: string;
  email: string;
}

export async function getUserById(id: number): Promise<User> {
  // Simulate async behavior (e.g., API call)
  return new Promise((resolve) =>
    setTimeout(() => {
      resolve({ id, name: 'Alice', email: 'alice@example.com' });
    }, 100)
  );
}
```

Here's how you would test it:

```
import { getUserById } from '../../src/services/userService';

test('getUserById returns user data', async () => {
  const user = await getUserById(1);
  expect(user).toEqual({
    id: 1,
    name: 'Alice',
    email: 'alice@example.com',
  });
});
```

22.3.2 Example 2: Using `.resolves` and `.rejects`

Jest provides a concise syntax using `.resolves` and `.rejects` to assert `Promise` results:

```
test('getUserById resolves with a user object', () => {
  return expect(getUserById(1)).resolves.toHaveProperty('email', 'alice@example.com');
});
```

> YES **Note**: You must `return` the `expect(...).resolves` or `expect(...).rejects` chain, or use `await`, to ensure Jest waits for the promise to settle.

22.3.3 Example 3: Testing Rejected Promises

Suppose the function throws if the user isn't found:

```
export async function getUserById(id: number): Promise<User> {
  if (id <= 0) {
    throw new Error('Invalid user ID');
  }

  return {
    id,
    name: 'Alice',
    email: 'alice@example.com',
  };
}
```

You can test this using `.rejects`:

```
test('getUserById throws error on invalid ID', async () => {
  await expect(getUserById(0)).rejects.toThrow('Invalid user ID');
});
```

22.3.4 Mocking Asynchronous Behavior

When testing code that depends on external async services, mock those services to avoid network calls.

Example: Mock an Async API

```
// File: src/api/userApi.ts
export async function fetchUser(id: number): Promise<User> {
  // Simulates real API
}
```

In your test:

```
import * as userApi from '../../src/api/userApi';

jest.mock('../../src/api/userApi');
```

```
const mockedFetchUser = userApi.fetchUser as jest.MockedFunction<typeof userApi.fetchUser>;

test('should return mocked user data', async () => {
  mockedFetchUser.mockResolvedValue({
    id: 1,
    name: 'Mocked User',
    email: 'mock@example.com',
  });

  const result = await userApi.fetchUser(1);
  expect(result.name).toBe('Mocked User');
});
```

22.3.5 Example 4: Handling Timeouts

Async code can involve delays (e.g., `setTimeout`, `setInterval`). You can simulate or skip real time using Jest's fake timers.

Example: Using Jest Fake Timers

```
// File: src/utils/delay.ts
export async function wait(ms: number): Promise<void> {
  return new Promise(resolve => setTimeout(resolve, ms));
}
```

Test it with fake timers:

```
import { wait } from '../../src/utils/delay';

jest.useFakeTimers();

test('wait resolves after timeout', async () => {
  const promise = wait(1000);

  jest.advanceTimersByTime(1000); // Fast-forward time
  await expect(promise).resolves.toBeUndefined();
});
```

Don't forget to reset timers after use:

```
afterEach(() => {
  jest.useRealTimers();
});
```

22.3.6 Catching Errors in Async Tests

For `try/catch` based error handling, test for thrown exceptions:

```
import { getUserById } from '../../src/services/userService';

test('should throw on invalid user ID using try/catch', async () => {
  try {
    await getUserById(-1);
    // Fail if no error is thrown
    fail('Expected error not thrown');
  } catch (e) {
    expect((e as Error).message).toBe('Invalid user ID');
  }
});
```

This pattern is useful when you need to inspect or handle the error object directly.

22.3.7 Summary

In this section, you learned how to:

- Test `async` functions using `async/await`, `.resolves`, and `.rejects`
- Mock async behavior for isolated tests
- Handle and assert on errors from rejected promises
- Use fake timers to test delayed or timeout logic

Testing asynchronous logic is essential for modern applications where delays, failures, and real-world timing can affect behavior. With the right tools and patterns, you can confidently test even the trickiest async code in TypeScript.

Chapter 23.

Best Practices for TypeScript

1. Clean Type Definitions
2. Avoiding `any`
3. Progressive Typing in Large Codebases
4. Code Style and Linting with `ESLint`

23 Best Practices for TypeScript

23.1 Clean Type Definitions

Clean, explicit, and reusable type definitions are the foundation of high-quality TypeScript code. They make your code easier to understand, maintain, and scale—especially in collaborative environments or large projects.

In this section, you'll learn:

- Why clean type definitions matter
- How to simplify complex types
- When to use interfaces vs type aliases
- How to avoid redundancy and document your types

23.1.1 Why Clean Types Matter

When type definitions are messy, overly generic, or repetitive, they can:

- Increase cognitive load for other developers (and your future self)
- Cause subtle bugs or inconsistencies
- Make refactoring more difficult
- Reduce the effectiveness of tooling like auto-completion and static analysis

Clean types offer the opposite:

YES Better **readability** YES Stronger **tooling support** YES Easier **collaboration** YES Safer **refactoring**

23.1.2 Simplifying Complex Types

Let's say you're working with a user object returned from an API:

Messy Approach

```
function createUser(user: {
  id: number;
  name: string;
  email: string;
  isAdmin: boolean;
}): void {
  // ...
}
```

This works, but it's verbose, hard to reuse, and doesn't convey intent clearly.

Clean Approach

```
interface User {
  id: number;
  name: string;
  email: string;
  isAdmin: boolean;
}

function createUser(user: User): void {
  // ...
}
```

YES **Cleaner**, **easier to reuse**, and **self-documenting**.

23.1.3 Reusing and Composing Types

Avoid repeating type structures across your codebase. Instead, define them once and reuse.

Example: Reusing Types in Multiple Functions

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

function getProductById(id: string): Product {
  // ...
}

function calculateDiscount(product: Product): number {
  // ...
}
```

YES If the `Product` shape changes, you only need to update it in one place.

You can also **compose** types to reduce duplication:

```
interface Timestamps {
  createdAt: Date;
  updatedAt: Date;
}

interface Order extends Timestamps {
  id: string;
  total: number;
}
```

23.1.4 Interfaces vs. Type Aliases

Both `interface` and `type` can define object shapes, but they have different strengths.

Feature	`interface`	`type`
Extending other types	YES yes	YES yes (with `&`)
Declaration merging	YES yes	NO no
Unions and primitives	NO no	YES yes
IDE preference (autocomplete)	YES often better	YES good too

Use `interface` when:

- Defining object structures, especially for classes or APIs
- You expect other developers to extend the type

```
interface Animal {
  name: string;
}

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

Use `type` when:

- Creating union types or more flexible compositions
- Combining primitives, tuples, or functions

```
type ID = string | number;

type Callback = (data: string) => void;
```

23.1.5 Avoiding Redundancy

Duplicate type definitions often lead to bugs. Prefer DRY (Don't Repeat Yourself) principles for types.

Dont do this:

```
function getUser(): { id: number; name: string } {
  return { id: 1, name: 'Alice' };
}

function showUser(user: { id: number; name: string }) {
  console.log(user.name);
}
```

Do this:

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

function getUser(): User {
  return { id: 1, name: 'Alice' };
}

function showUser(user: User) {
  console.log(user.name);
}
```

YES One definition = fewer bugs, better maintainability.

23.1.6 Documenting Types

Clear types are good—but **documented** types are even better.

You can add comments above your interfaces or fields to help other developers understand them:

```
/**
 * Represents a blog post.
 */
interface Post {
  /** Unique post ID */
  id: string;

  /** Title of the post */
  title: string;

  /** Post content in Markdown format */
  content: string;

  /** Publication timestamp */
  publishedAt: Date;
}
```

Many editors like VSCode will show this documentation during hover or autocomplete, which is especially helpful in large teams.

23.1.7 Summary

In this section, you learned how to write **clean, maintainable, and reusable** type definitions. Specifically:

- Clear type definitions make code easier to maintain and collaborate on

- Use `interface` and `type` thoughtfully, depending on the use case
- Avoid redundancy by reusing and composing types
- Document types to improve readability and tooling support

Clean types are more than just "correct"—they are **communicative**, **flexible**, and a sign of thoughtful engineering. In the next section, we'll dive into how to avoid the overuse of the `any` type, and what to use instead.

23.2 Avoiding `any`

The `any` type in TypeScript is both powerful and dangerous. It tells the compiler to **skip type checking** for that variable, essentially opting out of TypeScript's strongest feature: **type safety**.

While `any` can be useful in rare cases—such as when migrating JavaScript code—overusing it defeats the purpose of using TypeScript in the first place.

In this section, you'll learn:

- Why relying on `any` is risky
- Safer alternatives like `unknown` and type guards
- Strategies for minimizing `any` in real-world code
- How to incrementally improve typing in large or legacy projects

23.2.1 Why `any` Is Dangerous

When you use `any`, you're telling TypeScript:

> "Trust me, I know what I'm doing. Don't check this."

This disables all type checks and autocompletion for that value, which can lead to:

- **Runtime errors** that TypeScript could have prevented
- **Loss of tooling support** (e.g., IntelliSense and refactoring)
- **Hard-to-maintain code** as the application grows

Example: The Problem with `any`

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

getLength(42); //  Runtime error: 42 has no 'length' property
```

The compiler doesn't catch the error because `str` is `any`.

23.2.2 Use `unknown` Instead of `any`

If you truly don't know the type yet, prefer `unknown` over `any`.

- `unknown` is **safe**: it requires you to narrow the type before using it
- `any` is **unsafe**: it allows anything, including operations that might fail

Example: Using `unknown`

```
function processValue(value: unknown) {
  if (typeof value === 'string') {
    console.log(value.toUpperCase());
  } else {
    console.log('Not a string');
  }
}
```

YES TypeScript forces you to check the type before using it—no more silent failures.

23.2.3 Use Type Guards to Narrow `unknown`

Type guards are functions or conditions that check and narrow types at runtime.

Example: Type Guard for an Object Shape

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

function isUser(value: unknown): value is User {
  return (
    typeof value === 'object' &&
    value !== null &&
    'id' in value &&
    'name' in value
  );
}

function greet(value: unknown) {
  if (isUser(value)) {
    console.log(`Hello, ${value.name}`);
  } else {
    console.log('Not a user');
  }
}
```

YES TypeScript now knows `value` is a `User` inside the `if` block.

23.2.4 Gradually Replacing `any`

In legacy projects or early development stages, `any` might be used temporarily. Here's how to gradually replace it:

Use Type Annotations

Instead of:

```
const data: any = fetchData();
```

Try:

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

const data: Product = fetchData();
```

Use `as const` for literals

```
const config = {
  mode: 'production',
  retry: false,
} as const;
```

This locks down the types to the most specific values.

Use `typeof` and `ReturnType`

You can infer types from existing values or functions:

```
const user = { id: 1, name: 'Alice' };
type User = typeof user;

function getUser(): User {
  return user;
}
```

23.2.5 Better with `noImplicitAny`

Enable `noImplicitAny` in your `tsconfig.json` to catch untyped variables automatically:

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

This helps you detect missing type annotations early, encouraging you to write more complete types.

23.2.6 Example: From any to Type-Safe

Unchecked any

```
function handleInput(input: any) {
  console.log(input.toLowerCase());
}
```

Mistakenly calling `handleInput(123)` causes a runtime error.

Safe with Type Narrowing

```
function handleInput(input: unknown) {
  if (typeof input === 'string') {
    console.log(input.toLowerCase());
  } else {
    console.log('Invalid input');
  }
}
```

23.2.7 Summary

Avoiding `any` is one of the simplest and most effective ways to improve the safety and quality of your TypeScript code. In this section, you learned:

- Why `any` should be avoided
- How to use `unknown` and type guards for safer alternatives
- Strategies to incrementally replace `any` in your codebase
- How to enforce type safety with compiler options like `noImplicitAny`

TypeScript is most powerful when it can protect you from mistakes—so give it as much information as possible. In the next section, we'll explore **progressive typing** strategies for improving types in large or existing codebases.

23.3 Progressive Typing in Large Codebases

Adopting TypeScript in an existing JavaScript codebase—especially a large one—can feel overwhelming. However, TypeScript is designed to support **gradual adoption**, allowing teams to incrementally add types and gain benefits over time.

In this section, you'll learn strategies for progressively introducing TypeScript, including:

- Using `allowJs` and `checkJs` to mix JavaScript and TypeScript
- Adding types incrementally with JSDoc and type assertions
- Balancing speed and type safety during a staged migration

23.3.1 Why Migrate Gradually?

In large codebases, rewriting everything in TypeScript all at once is often **impractical and risky**. A better approach is to **adopt TypeScript one file, one module, or one feature at a time**.

Benefits of progressive typing:

YES Early access to TypeScript's benefits YES Lower risk and minimal disruption YES Easier collaboration in teams YES Better long-term maintainability

23.3.2 Step 1: Enable JavaScript Support

In `tsconfig.json`, you can configure TypeScript to work with `.js` files:

```
{
  "compilerOptions": {
    "allowJs": true,     // Let TypeScript process JavaScript files
    "checkJs": true      // Type-check JavaScript files
  },
  "include": ["src/**/*"]
}
```

- `allowJs`: Allows JS files in the project
- `checkJs`: Enables type-checking for JS files (optional, but helpful)

Now you can run the TypeScript compiler over your existing JavaScript code and get type errors and warnings—**without changing file extensions yet**.

23.3.3 Step 2: Use JSDoc Comments for Types

Without converting to `.ts`, you can start adding type information using **JSDoc annotations**:

Example: Adding JSDoc to JavaScript

```
/**
 * @param {string} name
 * @returns {string}
 */
function greet(name) {
  return `Hello, ${name}`;
}
```

TypeScript will infer the types and provide autocomplete, error checking, and refactoring tools.

You can even define complex types inline or import them:

```
/**
 * @typedef {Object} User
 * @property {number} id
 * @property {string} name
 */

/**
 * @param {User} user
 */
function printUser(user) {
  console.log(user.name);
}
```

YES This method gives you static type checking in JS without converting files right away.

23.3.4 Step 3: Rename Files Gradually to `.ts` or `.tsx`

After type-checking JavaScript files and fixing the most obvious issues, you can start converting files one at a time:

```
mv src/utils/math.js src/utils/math.ts
```

Or for React:

```
mv src/components/Button.jsx src/components/Button.tsx
```

TypeScript will now enforce full type-checking on the renamed file.

23.3.5 Step 4: Use Type Assertions When Needed

During migration, you might encounter untyped values. Instead of using `any`, use **type assertions** to tell TypeScript the expected type:

```
const data = JSON.parse(rawData) as User;
```

This tells TypeScript to treat `data` as a `User`, even if the compiler can't infer it directly.

Be cautious: assertions override the compiler's checks, so only use them when you're sure.

23.3.6 Step 5: Introduce Type Definitions Gradually

You don't need to type everything at once. Focus on:

1. **Public APIs** (functions, classes, exports)
2. **Common data models** (e.g. `User`, `Product`, `Order`)

3. **Frequently used utility functions**

As you convert files, you can extract types and interfaces into reusable type files (`types.ts`) and build from there.

23.3.7 Step 6: Use `// @ts-nocheck` Temporarily (Sparingly)

If you run into a file that's too messy to type immediately, you can silence errors temporarily:

```
// @ts-nocheck
```

This disables type checking in that file. It should be a **temporary escape hatch**, not a permanent solution. Prefer fixing type errors when possible.

23.3.8 Step 7: Use Tools and Linters to Track Progress

- `tsc --noEmit`: Run type checks without building files
- **ESLint with `@typescript-eslint`**: Helps enforce typing rules and catch bad patterns
- **ts-prune**: Identifies unused types or exports
- **Type Coverage Tools**: Tools like `type-coverage` or `typescript-coverage-report` help track how much of your project is typed

23.3.9 Balancing Speed and Safety

Migrating a large codebase takes time. Here's how to manage the balance:

Strategy	Speed YES	Safety YES
Enable `allowJs`	YES Fast	WARNING Basic type safety
Add JSDoc + `checkJs`	YES Fast	YES Safer types
Rename to `.ts`/`.tsx`	NO Slower	YES Full safety
Use assertions (`as Type`)	YES Handy	WARNING Use carefully
Skip-check with `@ts-nocheck`	YES Fast	NO No safety

Choose the right mix depending on the complexity and criticality of the module you're converting.

23.3.10 Summary

In this section, you learned how to **incrementally migrate** a large JavaScript codebase to TypeScript using:

- `allowJs` and `checkJs` for mixed codebases
- JSDoc comments to add types without file conversion
- Type assertions and selective file renaming
- Tools and strategies to balance speed with safety

With a progressive strategy, you can adopt TypeScript step-by-step—avoiding costly rewrites while still gaining all the benefits of type safety and better tooling.

In the next section, we'll look at maintaining consistent and readable code with **ESLint and TypeScript linting tools**.

23.4 Code Style and Linting with `ESLint`

Consistent code style and early error detection are essential for maintaining high-quality code—especially in team projects. **ESLint** is a popular, extensible tool that helps enforce coding standards and catch potential issues before they become bugs.

In this section, you'll learn:

- What ESLint is and why it matters for TypeScript
- How to configure ESLint with TypeScript support
- Common linting errors and how to fix them
- Benefits of using ESLint in team projects

23.4.1 What is ESLint?

ESLint is a static code analysis tool that checks your code for style issues and possible errors according to configurable rules. It helps you:

- Enforce consistent formatting and best practices
- Catch potential bugs like unused variables, unreachable code, or incorrect types
- Improve code readability and maintainability
- Integrate with IDEs for instant feedback

When combined with TypeScript, ESLint becomes even more powerful by understanding type information and enforcing type-aware rules.

23.4.2 Setting Up ESLint for TypeScript

Here's how to set up ESLint in a TypeScript project.

Step 1: Install Dependencies

```
npm install --save-dev eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin
```

- `eslint`: Core linter
- `@typescript-eslint/parser`: Parses TypeScript code so ESLint can analyze it
- `@typescript-eslint/eslint-plugin`: Adds TypeScript-specific linting rules

Step 2: Create `.eslintrc.js` Configuration

Create or update your `.eslintrc.js` file with the following:

```
module.exports = {
  parser: '@typescript-eslint/parser', // Specifies the ESLint parser for TS
  parserOptions: {
    ecmaVersion: 2020,                 // Allows modern ECMAScript features
    sourceType: 'module',              // Allows use of imports
  },
  plugins: ['@typescript-eslint'],
  extends: [
    'eslint:recommended',              // Basic recommended rules
    'plugin:@typescript-eslint/recommended', // TypeScript-specific recommended rules
  ],
  rules: {
    // Customize or override rules here
    '@typescript-eslint/no-unused-vars': ['error', { argsIgnorePattern: '^_' }],
    '@typescript-eslint/explicit-function-return-type': 'warn',
  },
};
```

Step 3: Add Lint Script to `package.json`

```
"scripts": {
  "lint": "eslint 'src/**/*.{ts,tsx}'"
}
```

Run linting with:

```
npm run lint
```

23.4.3 Common Linting Errors and Fixes

Here are some typical linting errors you might encounter, and how to address them:

Unused Variables

```
const unusedVar = 123;
// ESLint error: 'unusedVar' is defined but never used.
```

Fix: Remove unused variables or prefix them with _ if necessary.

```
const _unusedVar = 123;
```

Missing Return Types on Functions

```
function greet(name: string) {
  return `Hello, ${name}`;
}
// Warning: Missing explicit return type on function.
```

Fix: Add explicit return type:

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

Improper Use of any

```
function log(value: any) {
  console.log(value);
}
// Warning: Avoid using 'any' types.
```

Fix: Replace `any` with `unknown` or a specific type:

```
function log(value: unknown) {
  console.log(value);
}
```

Consistent Semicolons and Quotes

ESLint can enforce consistent use of semicolons and quotes:

```
const message = "Hello world"
// ESLint error: Missing semicolon
```

Fix: Add semicolon or configure rule for no semicolon style.

23.4.4 Benefits of Using ESLint in Teams

- **Consistency**: Everyone writes code with the same style, making reviews easier.
- **Early Bug Detection**: Catch issues like unused variables or type mismatches before runtime.
- **Better Onboarding**: New team members get immediate feedback about style expectations.
- **Improved Code Quality**: Encourages best practices and discourages bad habits.

Many editors like VSCode support ESLint natively, showing lint errors inline as you type, which accelerates development and reduces frustration.

23.4.5 Summary

In this section, you learned how to:

- Use ESLint to enforce consistent, readable code in TypeScript projects
- Configure ESLint with `@typescript-eslint` parser and plugin
- Identify and fix common linting errors
- Understand the benefits of linting for individual developers and teams

ESLint is a valuable tool that complements TypeScript's type system—helping you write clean, maintainable, and error-free code.

Chapter 24.

Common Pitfalls and Debugging Tips

1. Type-Related Errors and Solutions
2. Debugging with VSCode
3. Understanding Compiler Messages

24 Common Pitfalls and Debugging Tips

24.1 Type-Related Errors and Solutions

TypeScript's powerful static typing helps catch many errors at compile time, but beginners often encounter some common type-related errors that can be confusing at first. Understanding these errors and their solutions is crucial for becoming productive quickly.

In this section, we will explore frequent type errors such as:

- Incompatible types
- Null and undefined issues
- Excess property checks

For each, we'll explain why they occur and how to fix or work around them with clear, example-driven guidance.

24.1.1 Incompatible Types

Error Message Example:

```
Type 'number' is not assignable to type 'string'.
```

Cause: You're trying to assign a value of one type to a variable or parameter expecting a different type.

Example:

```
let name: string = 123; // Error: Type 'number' is not assignable to type 'string'.
```

Solution: Ensure the assigned value matches the declared type. You can convert types explicitly if necessary:

```
let name: string = String(123); // OK
```

Or adjust your types if your data allows multiple types, using a union:

```
let value: string | number = 123;
```

24.1.2 Null and Undefined Issues

Error Message Example:

```
Object is possibly 'null'.
```

or

```
Object is possibly 'undefined'.
```

Cause: TypeScript's strict null checking detects that a value might be `null` or `undefined`, but you're trying to access properties or methods without handling those cases.

Example:

```
function greet(name?: string) {
  console.log(name.toUpperCase()); // Error: Object is possibly 'undefined'.
}
```

Solution: Use **null checks** or **optional chaining**:

```
function greet(name?: string) {
  console.log(name?.toUpperCase() ?? 'Guest');
}
```

Or explicitly check:

```
function greet(name?: string) {
  if (name) {
    console.log(name.toUpperCase());
  } else {
    console.log('Guest');
  }
}
```

24.1.3 Excess Property Checks

Error Message Example:

```
Object literal may only specify known properties, and 'age' does not exist in type 'User'.
```

Cause: You're passing an object literal with extra properties to a function or variable expecting a specific type. TypeScript performs **excess property checks** on object literals to catch typos or unexpected fields.

Example:

```
interface User {
  name: string;
}

const user: User = { name: 'Alice', age: 30 }; // Error: 'age' does not exist in type 'User'.
```

Solution:

- Remove or correct the extra property.
- If you expect extra properties, use an index signature:

```
interface User {
  name: string;
  [key: string]: any; // Allows additional properties
}
```

- Or assign the object to a variable first to bypass excess checks:

```
const extra = { name: 'Alice', age: 30 };
const user: User = extra; // OK
```

24.1.4 Function Parameter Type Errors

Error Message Example:

```
Argument of type 'string' is not assignable to parameter of type 'number'.
```

Cause: Calling a function with an argument that does not match the declared parameter type.

Example:

```
function square(x: number) {
  return x * x;
}

square('5'); // Error
```

Solution: Pass the correct type:

```
square(5); // OK
```

Or update the function parameter type if it should accept multiple types:

```
function square(x: number | string) {
  const num = typeof x === 'string' ? Number(x) : x;
  return num * num;
}
```

24.1.5 Type Assertion Errors

Error Message Example:

```
Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps
```

Cause: You're asserting a type incorrectly. Type assertions tell the compiler to treat a value as a specific type, but invalid assertions cause errors.

Example:

```
const val = "123" as number; // Error
```

Solution: Use safe conversions instead of assertions:

```
const val = Number("123"); // Converts string to number safely
```

Only use assertions when you are sure about the type, such as when working with DOM

elements:

```
const input = document.getElementById('input') as HTMLInputElement;
```

24.1.6 Summary

Understanding these common type errors will help you debug issues faster and write better TypeScript code:

Error Type	Cause	Solution
Incompatible Types	Mismatched variable or parameter types	Match types or use unions
Null/Undefined Issues	Possible null/undefined values	Use null checks, optional chaining
Excess Property Checks	Extra fields in object literals	Remove extras, use index signatures, or assign separately
Function Parameter Errors	Arguments don't match parameter types	Pass correct types or widen parameter types
Type Assertion Errors	Unsafe or invalid assertions	Use proper conversions and only assert when sure

By recognizing and fixing these errors, you'll gain confidence in TypeScript's type system and speed up your development process.

24.2 Debugging with VSCode

Visual Studio Code (VSCode) is a popular code editor that offers excellent built-in debugging support for TypeScript projects. Using VSCode's debugging tools effectively can save you a lot of time by letting you inspect your running code, step through logic, and understand exactly what's happening at runtime.

In this section, you'll learn how to:

- Set up VSCode to debug TypeScript
- Configure breakpoints
- Inspect variables and call stacks
- Step through your code
- Use source maps to debug the original TypeScript, not just compiled JavaScript

24.2.1 Step 1: Configure Your TypeScript Project for Debugging

To debug TypeScript in VSCode, you need to make sure your project is configured to generate **source maps**, which map the compiled JavaScript back to your original TypeScript code.

In your `tsconfig.json`, enable source maps by adding:

```
{
  "compilerOptions": {
    "sourceMap": true,
    // other options...
  }
}
```

Source maps allow VSCode to link the running JavaScript code back to your `.ts` files.

24.2.2 Step 2: Create a Debug Configuration in VSCode

Next, set up a debug configuration by creating a `.vscode/launch.json` file in your project root.

For a Node.js TypeScript project, use this example:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}/dist/index.js",  // Adjust to your compiled JS entry point
      "preLaunchTask": "tsc: build - tsconfig.json",  // Compile TS before debugging
      "outFiles": ["${workspaceFolder}/dist/**/*.js"],
      "sourceMaps": true,
      "console": "integratedTerminal"
    }
  ]
}
```

This configuration:

- Launches the compiled JavaScript entry point (`dist/index.js`)
- Runs the TypeScript compiler before starting
- Enables source maps for debugging TypeScript files
- Uses the integrated terminal for output

24.2.3 Step 3: Set Breakpoints

Open any `.ts` file in VSCode, and click next to the line number to set a breakpoint (a red dot will appear).

When your debug session runs, execution will pause at the breakpoint, allowing you to inspect program state.

24.2.4 Step 4: Start Debugging

Press `F5` or click the green "Start Debugging" button in VSCode's debug panel to launch your program.

When execution hits a breakpoint, VSCode will switch to the debug view.

24.2.5 Step 5: Inspect Variables and Call Stack

While paused at a breakpoint, you can:

- **Hover over variables** in the editor to see their current values
- Use the **Variables panel** to inspect local, closure, and global variables
- Explore the **Call Stack panel** to see the sequence of function calls that led here
- Use the **Watch panel** to monitor specific expressions or variables

24.2.6 Step 6: Step Through Your Code

VSCode provides buttons to control execution:

- **Step Over (F10):** Execute the next line, skipping into function calls
- **Step Into (F11):** Dive into the next function call
- **Step Out (Shift+F11):** Exit the current function and return to the caller
- **Continue (F5):** Resume running until the next breakpoint or program end

Use these to follow the flow of your program carefully.

24.2.7 Example: Debugging a Simple TypeScript Application

Suppose you have this TypeScript code (`src/index.ts`):

```
function factorial(n: number): number {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}

console.log(factorial(5));
```

Your build script compiles to `dist/index.js` with source maps enabled.

1. Set a breakpoint on the line `return n * factorial(n - 1);`
2. Start the debug session in VSCode
3. When paused, hover over `n` to see its current value (starting at 5)
4. Use Step Into (F11) to see recursive calls unfold
5. Observe how the call stack grows and shrinks in the Call Stack panel
6. Use Step Over (F10) to execute lines without diving deeper

24.2.8 Practical Tips for Effective Debugging

- **Keep your source maps up to date:** Always recompile your code after changes to ensure debugging matches the latest code.
- **Use conditional breakpoints:** Right-click a breakpoint and add conditions to pause only when certain criteria are met.
- **Logpoints:** Instead of pausing, use logpoints to print messages without stopping execution.
- **Debug tests:** You can debug your Jest or other test runner tests by adding debug configurations.
- **Integrate with tasks:** Configure VSCode to compile TypeScript automatically before debugging.

24.2.9 Summary

VSCode's debugging tools, combined with source maps, let you step through your TypeScript code as if it were running natively—making it easier to understand bugs and logic errors.

You learned how to:

- Enable source maps in your project
- Create and configure a launch configuration for Node.js
- Set breakpoints and inspect variables
- Step through your code to follow execution
- Use practical tips like conditional breakpoints and logpoints

Mastering these debugging techniques will greatly improve your productivity and confidence

as a TypeScript developer.

24.3 Understanding Compiler Messages

One of the most important skills in learning TypeScript is being able to **interpret compiler error messages** effectively. These messages guide you to fix type errors and improve your code quality. Although TypeScript's errors can seem cryptic at first, understanding their patterns and codes will speed up your development and reduce frustration.

In this section, you'll learn how to:

- Read and interpret common TypeScript compiler errors
- Recognize typical error codes and patterns
- Use error messages to identify and fix issues efficiently

24.3.1 Anatomy of a Compiler Error Message

A typical TypeScript error message looks like this:

```
src/app.ts:10:5 - error TS2322: Type 'number' is not assignable to type 'string'.

10     let name: string = 42;
       ~~~~~~~~~~~~~~~~~~~~~
```

- **File and location**: `src/app.ts:10:5` tells you the file, line, and column of the error.
- **Error code**: `TS2322` is a unique identifier for this type of error.
- **Message**: `Type 'number' is not assignable to type 'string'.` describes the problem.
- **Code snippet**: The compiler shows the problematic code with a tilde underline.

24.3.2 Common Compiler Errors and How to Fix Them

TS2322 Type 'X' is not assignable to type 'Y'

What it means: You're assigning a value of one type to a variable or parameter expecting a different type.

Example:

```ts
let username: string = 123;  // Error TS2322
```

Fix: Make sure the value matches the variable type, or use a union type if multiple types

are allowed.

TS2532 Object is possibly 'null' or 'undefined'

What it means: You're trying to access a property or call a method on something that might be `null` or `undefined`.

Example:

```
function greet(name?: string) {
  console.log(name.toUpperCase());  // Error TS2532
}
```

Fix: Add null checks or use optional chaining:

```
console.log(name?.toUpperCase());
```

TS2345 Argument of type 'X' is not assignable to parameter of type 'Y'

What it means: You're passing an argument to a function that doesn't match the expected parameter type.

Example:

```
function square(n: number) { return n * n; }
square("5"); // Error TS2345
```

Fix: Pass the correct type or adjust the function parameter to accept more types.

TS2320 Interface X incorrectly extends interface Y

What it means: When extending interfaces, the new interface doesn't correctly implement all required properties or method types.

Example:

```
interface Animal {
  name: string;
}
interface Dog extends Animal {
  age: number;
}
const dog: Dog = { name: "Rex" };  // Error TS2320: missing 'age'
```

Fix: Add missing properties or correct their types.

TS2554 Expected X arguments, but got Y

What it means: You called a function with the wrong number of arguments.

Example:

```
function add(a: number, b: number) { return a + b; }
add(5); // Error TS2554
```

Fix: Provide all required arguments or make some optional.

24.3.3 Using Error Codes to Find Help

The `TS` error codes are consistent and documented. You can search online for:

`TS2322 Type 'number' is not assignable to type 'string'`

and find detailed explanations and community discussions. The official TypeScript GitHub issues and Microsoft Docs often provide examples and solutions.

24.3.4 Tips for Quickly Diagnosing Errors

- **Check the line and column** mentioned at the top of the error message.
- **Read the message carefully**; it often states what type was expected vs. what was found.
- **Look for underlined code** in your editor showing the exact error spot.
- **Use type annotations and explicit types** to make errors clearer.
- **Incrementally fix errors**; sometimes fixing one error resolves many others.

24.3.5 Summary

TypeScript compiler messages are your best friends for writing correct code. In this section, you learned:

- How to read error messages, including file location, error code, and description
- Common error codes like TS2322, TS2532, and TS2345 and what they mean
- Strategies for using the error output to quickly identify and fix issues

Mastering these messages will help you write more robust TypeScript code and reduce time spent debugging.