**TypeScript Best Practices: Writing Clean and Modern Code**

[[Evelyn Taylor](https://medium.com/@Evelyn.Taylor?source=post_page-----a04196310e4d--------------------------------)](https://medium.com/@Evelyn.Taylor?source=post_page-----a04196310e4d--------------------------------)

[[Stackademic](https://blog.stackademic.com/?source=post_page-----a04196310e4d--------------------------------)](https://blog.stackademic.com/?source=post_page-----a04196310e4d--------------------------------)

[Evelyn Taylor](https://medium.com/@Evelyn.Taylor?source=post_page-----a04196310e4d--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ffd2770b746ea&operation=register&redirect=https%3A%2F%2Fblog.stackademic.com%2Ftypescript-best-practices-writing-clean-and-modern-code-a04196310e4d&user=Evelyn+Taylor&userId=fd2770b746ea&source=post_page-fd2770b746ea----a04196310e4d---------------------post_header-----------)

Published in

[Stackademic](https://blog.stackademic.com/?source=post_page-----a04196310e4d--------------------------------)

·

7 min read

·

Sep 14

152

3



Photo by [Christopher Gower](https://unsplash.com/@cgower?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

So, you’ve decided to dive headfirst into the world of TypeScript. Great choice! TypeScript has come a long way, and it’s a powerful tool for building robust, maintainable web applications.

Whether you’re a seasoned developer looking to level up your skills or a newbie eager to learn, this guide will walk you through some TypeScript best practices for 2023.

We’ll keep things casual, provide real-world examples, and introduce you to some modern techniques that will make your TypeScript code shine.

**What is TypeScript, Anyway?**

Before we dive into best practices, let’s briefly recap what TypeScript is.

*TypeScript is a statically typed superset of JavaScript that adds type annotations to your code.*

It helps you catch errors early in development, improves code quality, and enhances code maintainability. Think of it as JavaScript with superpowers!

Now, let’s get into the nitty-gritty of writing clean and modern TypeScript code.

**1. TypeScript’s Strict Mode**

TypeScript’s strict mode is a compiler option that enforces stricter type checking and better coding practices to catch potential errors at compile-time rather than runtime.

It helps you write safer and more maintainable code by reducing common programming mistakes.

To enable strict mode, you can set the "strict" compiler option to true in your TypeScript configuration file (tsconfig.json).

Here’s an example that demonstrates TypeScript’s strict mode with some code snippets:

**Enabling Strict Mode:**

In your tsconfig.json file, set "strict" to true or specify individual strict options:

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

Alternatively, you can enable specific strict flags like "strictNullChecks", "strictFunctionTypes", "strictPropertyInitialization", etc., individually based on your needs.

1. **Strict Null Checks:**

Strict Null Checks in TypeScript help catch potential issues related to null and undefined values at compile-time, reducing runtime errors.

{  
 "compilerOptions": {  
 "strictNullChecks": true  
 }  
}

Here's an example illustrating how strict null checks work:

// Without strict null checking  
function getProperty(obj: any) {  
 return obj.myProperty; // No compile-time error, but may result in runtime error if obj is null or undefined  
}  
  
// With strict null checking  
function getProperty(obj: any) {  
 return obj.myProperty; // Compile-time error: Object is possibly 'undefined' or 'null'  
}  
  
// Using strict null checking properly  
function getProperty(obj: { myProperty?: string }) {  
 return obj.myProperty; // No compile-time error because we've specified that myProperty is optional  
}  
  
// Example usage  
const obj1 = { myProperty: "Hello" };  
const obj2 = null;  
  
console.log(getProperty(obj1)); // Outputs: "Hello"  
console.log(getProperty(obj2)); // Compile-time error: Argument of type 'null' is not assignable to parameter of type '{ myProperty?: string | undefined; }'

In the example above:

1. In the first example without strict null checking, TypeScript allows accessing obj.myProperty even if obj is null or undefined. This could lead to runtime errors if obj is not properly checked before usage.
2. In the second example with strict null checking enabled, TypeScript detects the potential issue and flags it as a compile-time error.
3. In the third example, we specify that myProperty is optional using myProperty?: string. This tells TypeScript that it's okay for myProperty to be missing, so there are no compile-time errors in this case.

**2. Strict Function Types:**

Strict function types, also known as strong or statically-typed function types, are a concept in programming languages where the types of function parameters and return values are explicitly defined and enforced by the compiler or interpreter.

This means that the types of arguments passed to a function and the type of the value returned by the function must match the declared types, and any type mismatches are caught at compile-time rather than runtime.

Here’s an example in a statically-typed language like TypeScript:

function add(a: number, b: number): number {  
 return a + b;  
}  
  
const result = add(3, 5); // This works fine  
const invalidResult = add("Hello", 5); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.

In this example:

1. We define a function add that takes two parameters, a and b, both of type number, and returns a value of type number.
2. When we call add(3, 5), it works fine because we are passing two numbers as arguments, and the function returns a number.
3. However, when we try to call add("Hello", 5), we get a compile-time error because we are trying to pass a string ("Hello") as the first argument, which is not compatible with the function's parameter type of number.

**2. Explicitly Type Your Variables**

In TypeScript, explicitly typing your variables is considered a best practice as it helps catch type-related errors during development and provides better code documentation.

TypeScript provides several ways to specify types for variables. Here are some examples of how to explicitly type variables in TypeScript:

1. **Primitive Types:**

// Explicitly typed variable with a string type  
const username: string = "John Doe";  
  
// Explicitly typed variable with a number type  
const age: number = 30;  
  
// Explicitly typed variable with a boolean type  
const isActive: boolean = true;

**2. Arrays:**

// Explicitly typed array of numbers  
const numbers: number[] = [1, 2, 3, 4, 5];  
  
// Explicitly typed array of strings  
const fruits: string[] = ["apple", "banana", "cherry"];

**3. Objects**

// Explicitly typed object with properties  
const person: { name: string; age: number } = {  
 name: "Alice",  
 age: 25,  
};

**4. Custom Types (Interfaces or Types):**

// Using an interface to define the type  
interface Point {  
 x: number;  
 y: number;  
}  
  
const point: Point = { x: 10, y: 20 };

**5. Function Signatures:**

// Explicitly typed function parameter and return type  
function add(a: number, b: number): number {  
 return a + b;  
}

**6. Union Types:**

// Explicitly typed variable with a union type  
const result: string | number = Math.random() > 0.5 ? "Hello" : 42;

**7. Type Assertions:**

You can use type assertions when TypeScript cannot infer the type, but you are sure of it:

const element = document.getElementById("myElement");  
  
// Asserting that 'element' is an HTMLInputElement  
const inputElement = element as HTMLInputElement;

**3. Use Interfaces for Objects**

Using interfaces for objects in TypeScript is considered a best practice because it enhances code readability, maintainability, and type checking.

Interfaces provide a way to define the shape or structure of objects, making it clear what properties an object should have.

Here are some reasons why using interfaces for objects is a best practice:

1. **Readability and Documentation:** Interfaces serve as a form of documentation for your code. They clearly specify the expected properties and their types, making it easier for other developers (or even your future self) to understand the intended structure of objects.
2. **Type Checking:** TypeScript enforces that objects that claim to implement an interface must have all the properties and methods specified in that interface. This helps catch errors at compile-time rather than runtime, reducing the chances of bugs in your code.
3. **Code Reusability:**You can use interfaces to define a contract that multiple objects or classes can adhere to. This promotes code reusability and helps ensure that different parts of your codebase work consistently with objects of the same shape.
4. **Intellisense and IDE Support:**Modern code editors and IDEs provide excellent support for TypeScript. Using interfaces enhances autocompletion and provides type hints as you work with objects in your code, improving developer productivity.

Here’s an example of how to use interfaces for objects in TypeScript:

// Define an interface for a User object  
interface User {  
 id: number;  
 username: string;  
 email: string;  
 // Optional properties can be defined with '?'  
 age?: number;  
}  
  
// Create an object that adheres to the User interface  
const newUser: User = {  
 id: 1,  
 username: "john\_doe",  
 email: "john@example.com",  
};  
  
// TypeScript enforces that newUser has the required properties of the User interface  
// Adding or removing properties would result in a compilation error

**4. Avoid the**any**Type**

The any type should be used sparingly, if at all. It's a bit like saying, "I give up, TypeScript, you figure it out."

// Bad  
let data: any = fetchDataFromAPI();  
  
// Good  
let data: MyDataType = fetchDataFromAPI();

any weakens TypeScript's strong type-checking capabilities. Be explicit about your types to catch potential issues.

**5. Use Enums for Constants**

Instead of using magic strings or numbers, use enums to define constants in your code. It makes your code more readable and maintainable.

// Bad  
const status = "success";  
  
// Good  
enum Status {  
 Success = "success",  
 Error = "error",  
}  
  
const status = Status.Success;

Enums provide a clear and self-documenting way to represent a fixed set of values.

**6. Leverage Union and Intersection Types**

Union and intersection types are powerful tools in TypeScript. Union types allow you to work with values that can be one of several types, while intersection types combine multiple types.

// Union Type  
function formatInput(input: string | number): string {  
 return `You entered: ${input}`;  
}  
  
// Intersection Type  
interface Named {  
 name: string;  
}  
  
interface Aged {  
 age: number;  
}  
  
type Person = Named & Aged;

These types help you express complex data structures more accurately.

**7. Utilize Type Inference**

Let TypeScript do some of the heavy lifting. If you define your function’s return type, TypeScript can often infer the types of its parameters.

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

It keeps your code DRY (Don’t Repeat Yourself) and reduces redundancy.

**8. TSLint or ESLint with TypeScript**

Linters like TSLint or ESLint with TypeScript plugins can help you catch style issues, potential bugs, and enforce coding standards.

Consistency in code style and error prevention make your codebase more maintainable.

**9. Keep Dependencies Typed**

When using third-party libraries, ensure you have type definitions for them. If not, you can write your own or use libraries like @types/xyz for popular packages.

It prevents type-related errors when working with external libraries.

**10. Write Meaningful Comments**

Last but not least, don’t underestimate the power of good comments. Explain your intentions, assumptions, and any complex logic.

Well-documented code is easier to maintain, especially when others need to dive into it.

**Conclusion:**

In conclusion, TypeScript is a fantastic tool to make your JavaScript code more robust.

Following these TypeScript best practices will save you headaches down the road and make your codebase more maintainable and enjoyable to work with.

So, go ahead, write some clean TypeScript code, and keep those bugs at bay!