

# **TYPESCRIPT HANDBOOK**



FLAVIO COPES

# Preface

This book aims to introduce you to TypeScript, a superset (or “version”) of JavaScript that’s used by most companies working with this technology.

JavaScript is the most popular programming language of the world. It powers every single website you visit.

If you’re new to JavaScript, read my [JavaScript Handbook](#) first.

TypeScript provides a set of features built on top of JavaScript that greatly improve the DX (Developer Experience) by adding static typing.

Variable types are checked at compile-time before the code is run, which helps catch type-related errors early in the development process.

This greatly improves the experience of working within an editor, as the editor is now equipped with more information about your code and can help you prevent errors, with the help of TypeScript’s tooling.

Many of those errors that the editor can highlight couldn’t be detected with plain JavaScript.

This is not the only feature of TypeScript, but to me it’s the best one.

TypeScript provides many language features over JavaScript, while still maintaining backwards compatibility (valid JavaScript is also valid TypeScript).

In this handbook we’ll talk about many of those features, in a way that’s easy to understand as a novice.

You’ll learn the basic concepts in a very straightforward way, without too much jargon or complicated explanations, using an 80/20 approach that gives you the mental models and knowledge to start using TypeScript.

The book was updated late 2025.

## Legal

Flavio Copes, 2025. All rights reserved.

Downloaded from [flaviocopes.com](https://flaviocopes.com).

No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher.

The information in this book is for educational and informational purposes only and is not intended as legal, financial, or other professional advice. The author and publisher make no

representations as to the accuracy, completeness, suitability, or validity of any information in this book and will not be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its use.

This book is provided free of charge to the newsletter subscribers of Flavio Copes. It is for personal use only. Redistribution, resale, or any commercial use of this book or any portion of it is strictly prohibited without the prior written permission of the author.

If you wish to share a portion of this book, please provide proper attribution by crediting Flavio Copes and including a link to [flaviocopes.com](https://flaviocopes.com).

## Introduction

We can't be Web Developers in this day and age without knowing TypeScript.

Few technologies in the last few years had the impact that TypeScript had over JavaScript and Web Developers in general.

Its best feature in my opinion is adding types and allowing the tools you work with to enforce those types, and help you write better code.

You are likely required to use it in a future project, or at your next job.

TypeScript knowledge will definitely help you get a job too, so let's just dive into it.

## Your first TypeScript program

TypeScript is a programming language that compiles to JavaScript.

You've heard it right.

Browsers can only execute JavaScript, and you can't tell Chrome "run this TypeScript code".

So you need a *compilation step* before being able to run TypeScript.

Most of the time, this compilation step is done by tools you already use, like Vite or Next.js or Astro or whatever. Maybe your server runtime too, if you use Deno or Bun.

You write TypeScript in `.ts` files.

Your first TypeScript file can easily be "just JavaScript":

```
const greet = () => {  
  console.log('Hello world!')  
}  
  
greet()
```

TypeScript is a **superset** of JavaScript.

This means that in theory *any JavaScript is valid TypeScript*.

But of course we want it to do *more*, otherwise ... why use it at all?

And the thing you do with TypeScript is adding types.

## Types

Typing is the key TypeScript feature.

So far we compiled a `.ts` file, but we just compiled plain JavaScript.

The most important piece of functionality provided by TypeScript is the type system.

If you ever used a typed language, like Go or C, you already know how this works. If not, and you only programmed in a dynamic language like Python or Ruby, this is all new to you but don't worry.

The type system allows you, for example, to add types to your variables, function arguments and function return types, giving a more rigid structure to your programs.

The advantages are better tooling: the compiler (and editors like [VS Code](#)) can help you a lot during development, pointing out bugs as you write the code. Bugs that couldn't possibly be detected if you didn't have types. Also, teamwork gets easier because the code is more explicit.

The resulting JavaScript code that we compile to does not have types, of course: they are lost during the compilation phase, but the compiler will point out any error it finds.

Here is how you define a string variable in TypeScript:

```
const greeting : string = 'hello!'
```

*Type inference* lets us avoid writing the type in obvious cases like this:

```
const greeting = 'hello!'
```

The type is determined by TS.

So basically our TypeScript can look like JavaScript. But each variable is now typed, and the editor (and the compiler) can help us write more correct code.

## Typing functions

A function can accept values as parameters, and return values after its execution.

So we must have 2 types: the type of the parameters values, and the type of the return value.

This is how a function accepts arguments of a specific type:

```
const multiply = (a: number, b: number) => {  
  return a * b  
}
```

Now try writing `multiply` in your editor, you will see VS Code suggests you the types of the function as you type it:

```
const multiply = (a: number, b: number) => {  
  return a * b  
}
```

mul

multiply	>
module	
Module	node:module
Module	node:module
Module	node:vm
mainModule?	node:process
isModuleNamespaceObject	node:util/types

const multiply: (a: number, b: number) => number

```
const multiply = (a: number, b: number) => {  
  return a * b  
}
```



multiply(a: number, b: number): number

multiply()

```
const multiply = (a: number, b: number) => {  
  return a * b  
}  
💡 multiply(a: number, b: number): number  
multiply(4, )
```

This is really handy especially when using functions you didn't write, maybe coming from a library you imported.

If you watch closely, TypeScript already inferred the return type is `number`.

In this case you don't need to, but here is how functions can *explicitly declare* their return value:

```
const multiply = (a: number, b: number): number => {  
  return a * b  
}
```

## The editor helps you with type errors

If you pass a string to `multiply()`, VS Code will show you an error by underlining the problematic part of your code:

```
const multiply = (a: number, b: number) => {  
  return a * b  
}  
  
multiply(4, 'test')
```

Hover that with the mouse, VS Code will tell you more:

```
const multiply = (a: number, b: number) => {  
  return a * b  
}  
  
multiply(4, 'test')
```

Argument of type 'string' is not assignable to parameter of type 'number'. ts(2345)

△ Error (TS2345) [x] [⊞]

Argument of type **string** is not assignable to parameter of type **number** .

View Problem (⌘F8) No quick fixes available

Now, regarding TypeScript errors, sometimes they are a bit cryptic. Not in this case. But there is a VS Code extension that helps you by giving actionable information, and it's called [Pretty TypeScript Errors](#).

Install that, you'll see different error messages:

```
const multiply = (a: number, b: number) => {  
  return a * b  
}  
  
multiply(4, 'test')
```

Argument of type 'string' is not assignable to parameter of type 'number'. ts(2345)

View Problem (⌘F8) No quick fixes available

This is what happens in the editor.

## Running TypeScript code

Now let's try running this code.

I'll use [Bun](#), which has built-in support for TypeScript.

Write this in a `test.ts` file:

```
const multiply = (a: number, b: number): number => {  
  return a * b  
}  
  
console.log(multiply(4, 2))
```

Run `bun test.ts` and you'll see the output `8` in the terminal:

Note: Running TypeScript directly requires a runtime with built-in TS support (like Bun or Deno) or a tool like ts-node. Otherwise, you need to compile to JavaScript first.

```
● → tstest bun test.js
8
○ → tstest █
```

What if we had a type error in our code?

Bun will happily run the code, as if it was JavaScript.

TS test.ts > ...

```
1  const multiply = (a: number, b: number): number => {
2    return a * b
3  }
4
5  console.log(multiply(4, {}))
6
```

TERMINAL

fish + v

```
● → tstest bun test.ts
NaN
○ → tstest █
```

This is because Bun does not type check before execution.

To do that, we must use the TypeScript compiler.

Install `typescript` in the project:

```
bun i -D typescript
```

Then run

```
npx tsc test.ts
```



```
⊗ → tstest bun tsc test.ts
test.ts:5:25 - error TS2345: Argument of type '{}' is not assignable
to parameter of type 'number'.
```

```
5 console.log(multiply(4, {}))
                        ~~
```

```
Found 1 error in test.ts:5
```

tsc is the TypeScript compiler.

It's up to you now to go and fix the problem, and recompile:

```
TS test.ts > ...
```

```
1  const multiply = (a: number, b: number): number => {
2    |   return a * b
3  }
4
5  console.log(multiply(4, 3))
6
```

```
TERMINAL
```

```
● → tstest bun tsc test.ts
○ → tstest
```

The compilation created a corresponding `.js` file that is the compiled JavaScript that browsers will be able to understand:

```
JS test.js > ...
```

```
1  var multiply = function (a, b) {  
2    return a * b;  
3  };  
4  console.log(multiply(4, 3));  
5
```

## Valid types

I've introduced a few types so far.

Valid types are

- `number`
- `string`
- `boolean`
- `void`
- `null`
- `undefined`
- `Array`
- ...and more.

Then we have `any`, a catch-all type that allows any type, which is special in the sense we shouldn't use it, as it removes many benefits of type checking, but sometimes it's an "easy way out" from having to define types in a strict way.

Prefer `unknown` over `any` when you don't know the type but want type safety. You'll need to check the type before using it:

```
const data: unknown = fetchData()  
if (typeof data === 'string') {  
  console.log(data.toUpperCase()) // now TS knows it's a string  
}
```

## Type aliases and interfaces

We've seen types for basic values.

For more complex structures we can use type aliases:

```
type Dog = {  
  name: string  
  age: number  
}
```

Then when you create an object, you set this to be its type:

```
const jack: Dog = {  
  name: 'Jack',  
  age: 3  
}
```

Note that we can have an optional property using `?:` when defining the type, like this:

```
type Dog = {  
  name: string  
  age?: number //optional  
}
```

To do the same thing we can also use *interfaces*:

```
interface Dog {  
  name: string  
  age: number  
}
```

They work in the same way as type aliases:

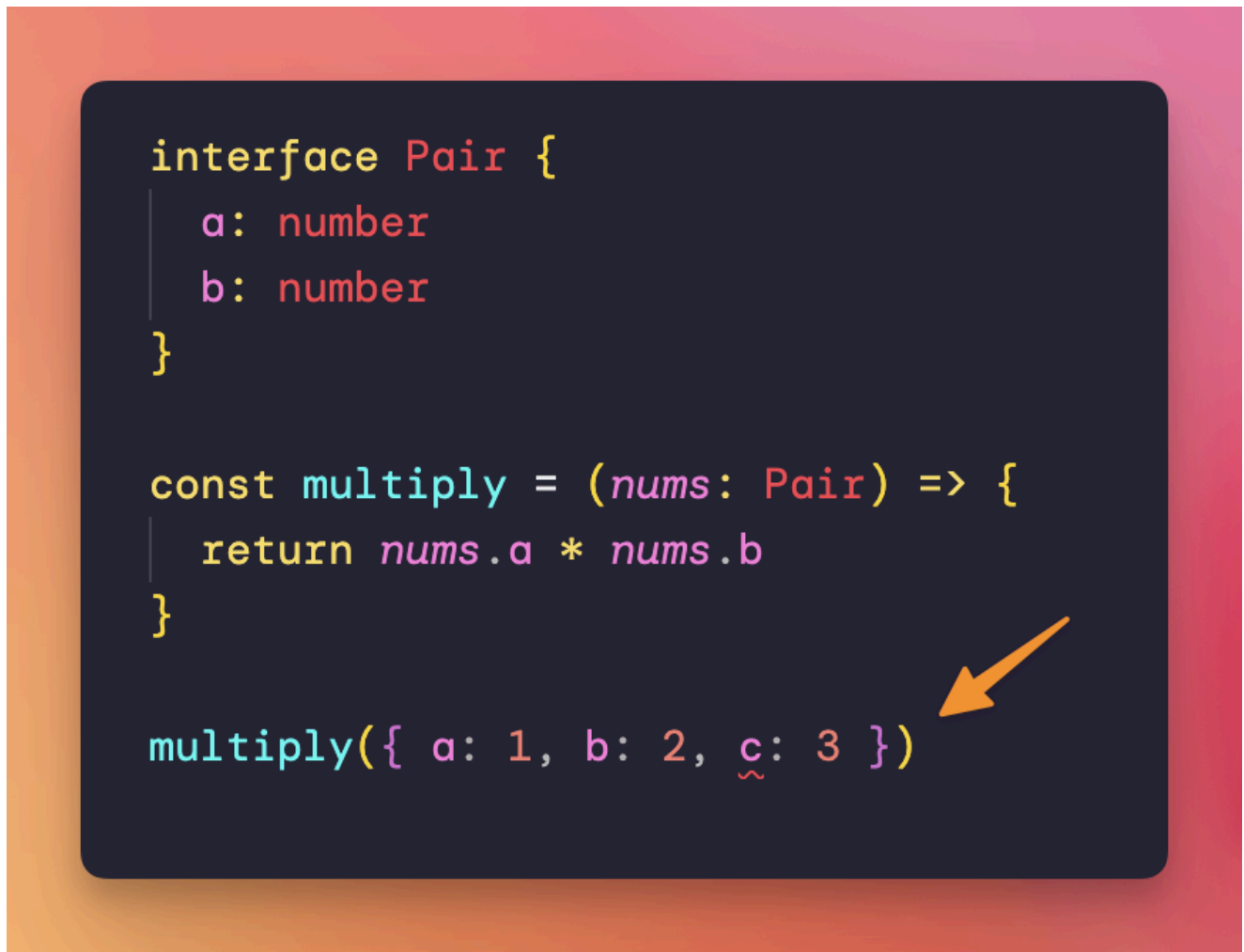
```
const jack: Dog = {  
  name: 'Jack',  
  age: 3  
}
```

Why use one vs another? There are [a few differences](#), but generally you can use one, or another, no big deal.

Type aliases or interfaces are not limited to typing objects, of course:

```
interface Pair {  
  a: number  
  b: number  
}  
  
const multiply = (nums: Pair) => {  
  return nums.a * nums.b  
}  
  
multiply({ a: 1, b: 2 })
```

If you pass a third parameter, TS will raise an error:



```
interface Pair {  
  a: number  
  b: number  
}  
  
const multiply = (nums: Pair) => {  
  return nums.a * nums.b  
}  
  
multiply({ a: 1, b: 2, c: 3 })
```

"Object literal may only specify known properties, and `c` does not exist in type `Pair`"


## Union types

Union types let us tell TypeScript a value can be of one type, or another type.

Example:

```
const run = (a: number | string) => {  
  //...  
}
```

In this case the parameter `a` can be either a string, or a number, but nothing else.



```
const run = (a: number | string) => {  
  //...  
}  
  
run(2)  
run('test')  
run({})
```

## Optional chaining and nullish coalescing

TypeScript supports modern JavaScript features like optional chaining ( `?.` ) and nullish coalescing ( `??` ):

```
const user = {  
  profile?: {  
    name?: string  
  }  
}  
  
const name = user?.profile?.name ?? 'Anonymous'
```

This safely accesses nested properties and provides a fallback value.

## Type guards

You can use type guards to narrow types at runtime:

```
function isString(value: unknown): value is string {  
  return typeof value === 'string'  
}
```

```

}

const data: unknown = 'hello'
if (isString(data)) {
  console.log(data.toUpperCase()) // TS knows data is string here
}

```

The `typeof` operator also works as a type guard in TypeScript.

## The `as const` assertion

Use `as const` to get literal types instead of general types:

```

const colors = ['red', 'green', 'blue'] as const
// Type: readonly ['red', 'green', 'blue'] instead of string[]

const config = {
  apiUrl: 'https://api.example.com',
  timeout: 5000
} as const
// Properties are readonly and have literal types

```

## Basic generics

Generics let you create reusable components that work with different types:

```

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

const num = identity<number>(42) // returns number
const str = identity<string>('hello') // returns string

```

You'll see generics everywhere in TypeScript, especially with `Promise<T>`, `Array<T>`, and React's `useState<T>()`.

## Typing arrays with generics

To type arrays, we use generics.

Suppose you want an array to only contain numbers.

This is how you type it:

```
const nums: number[] = [1, 2, 3]
```

You can now add numbers to the array, but if you add a string for example you'll have an error:

```
const nums: Array<number> = [1, 2, 3]
```

```
nums.push('test')
```

Argument of type 'string' is not assignable to parameter of type 'number'. ts(2345)

⚠ Error (TS2345) [?] | 🌐

Argument of type `string` is not assignable to parameter of type `number`.

[View Problem](#) (⌘F8) No quick fixes available

We use generics often with `useState` in React:

```
type GitHubData = {
  avatar_url: string
  name: string
  created_at: string
  bio: string
}

function App() {
  const [data, setData] = useState<GitHubData>()
}
```

## The DX of editing TypeScript

I think the key aspect of using TypeScript is in the developer's experience (DX) that it gives us.

The end result of TypeScript will be JavaScript without types.

Because the browser cannot understand types (yet?), they are stripped away once the program is compiled to JS.

So what are they for?

They are for the editing experience.

While working on your app in the editor, the editor will be able to automatically understand if the program has an error, much more than what it can do with JavaScript.

So you can catch possible sources of problems before even running your code.

And, as an additional benefit, when you're writing code, the editor (VS Code for example, but others too if set up for this) will give you hints about the types, the variables and parameters that you should write.

And you commonly get this "for free" when using libraries that provide types.

TypeScript is not the only way to do this.

In plain JavaScript we have [JSDoc](#), and many libraries prefer using this.

What matters is that our life is easier either way.

You can decide to use JavaScript, but the benefits using other libraries are the same.

And if you choose to use TypeScript, you can benefit from having types in your code, something that I think makes sense when your app grows larger, and you are working with a team of people.

When you can't keep all the code in your head any more.

TypeScript gives you more structure and things to worry about. It forces you to think about types, and sometimes for simple projects you might find it gets in the way.

JavaScript is great because it's dynamic and gives you a lot of freedom. But freedom can be a double-edged sword, because it's a bit too permissive.

But we're lucky enough to be able to choose between those 2 "versions" of JavaScript. You can pick the one you like the most, depending on the situation you're facing.

## There's more...

TypeScript doesn't end here, of course.

But TS is a whole programming language, and there are a ton of features that you can use once you choose to use it and go all-in.

I personally think the best way to use TypeScript is to just use its type system, so we basically write JavaScript with types, and ignore all the other TypeScript features that are built to help you write more structured programs, for example using classes and more complex inheritance.



I find more joy in writing more function-oriented code rather than going all-in on OOP, so I tend to use TypeScript for its types only.

Your mileage might vary, and your team might require you to use more TS features.

If you want to explore more of what TypeScript can offer, go ahead to the [official docs](#) to learn all the details, or start writing your apps and learn as you do!