Chapter 2. TypeScript: A 10_000 Foot View

Over the next few chapters, I'll introduce the TypeScript language, give you an overview of how the TypeScript Compiler (TSC) works, and take you on a tour of TypeScript's features and the patterns you can develop with them. We'll start with the compiler.

The Compiler

Depending on what programming languages you worked with in the past (that is, before you decided to buy this book and commit to a life of type safety), you'll have a different understanding of how programs work. The way TypeScript works is unusual compared to other mainstream languages like JavaScript or Java, so it's important that we're on the same page before we go any further.

Let's start broad: programs are files that contain a bunch of text written by you, the programmer. That text is parsed by a special program called a *compiler*, which transforms it into an *abstract syntax tree (AST)*, a data structure that ignores things like whitespace, comments, and where you stand on the tabs versus spaces debate. The compiler then converts that AST to a lower-level representation called *bytecode*. You can feed that bytecode into another program called a *runtime* to evaluate it and get a result. So when you run a program, what you're really doing is telling the runtime to evaluate the bytecode generated by the compiler from the AST parsed from your source code. The details vary, but for most languages this is an accurate high-level view.

Once again, the steps are:

1. Program is parsed into an AST.
2. AST is compiled to bytecode.
3. Bytecode is evaluated by the runtime.

Where TypeScript is special is that instead of compiling straight to bytecode, TypeScript compiles to… JavaScript code! You then run that JavaScript code like you normally would—in your browser, or with NodeJS, or by hand with a paper and pen (for anyone reading this after the machine uprising has begun).

At this point you may be thinking: "Wait! In the last chapter you said TypeScript makes my code safer! When does that happen?"

Great question. I actually skipped over a crucial step: after the TypeScript Compiler generates an AST for your program—but before it emits code—it *typechecks* your code.

TYPECHECKER

A special program that verifies that your code is typesafe.

This typechecking is the magic behind TypeScript. It's how TypeScript makes sure that your program works as you expect, that there aren't obvious mistakes, and that the cute barista across the street really will call you back when they said they would. (Don't worry, they're probably just busy.)

So if we include typechecking and JavaScript emission, the process of compiling TypeScript now looks roughly like Figure 2-1:

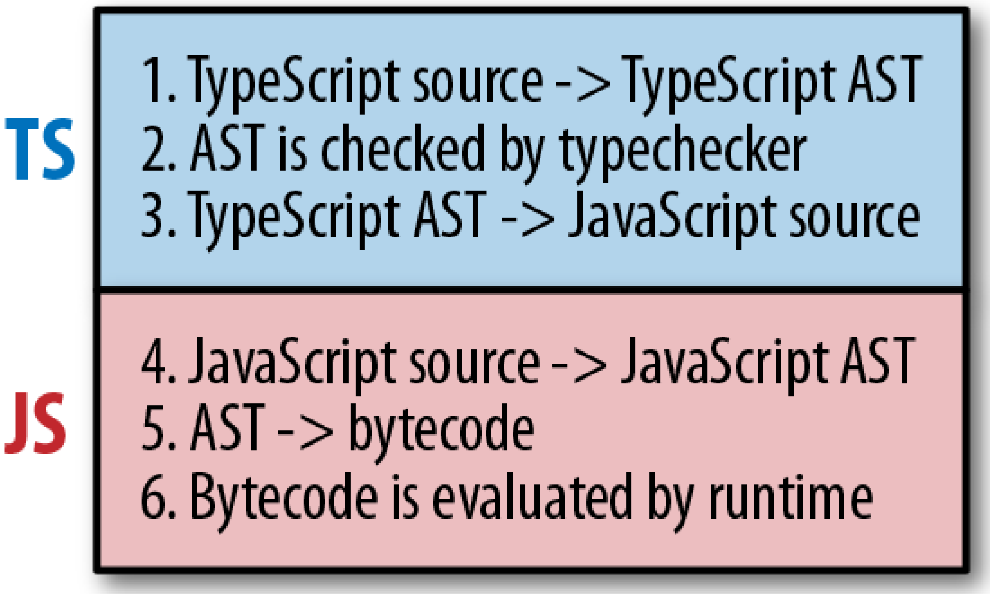

TS
1. TypeScript source -> TypeScript AST
2. AST is checked by typechecker
3. TypeScript AST -> JavaScript source
JS
4. JavaScript source -> JavaScript AST
5. AST -> bytecode
6. Bytecode is evaluated by runtime

Figure 2-1. Compiling and running TypeScript

Steps 1–3 are done by TSC, and steps 4–6 are done by the JavaScript runtime that lives in your browser, NodeJS, or whatever JavaScript engine you're using.

NOTE

JavaScript compilers and runtimes tend to be smushed into a single program called an *engine*; as a programmer, this is what you'll normally interact with. It's how V8 (the engine powering NodeJS, Chrome, and Opera), SpiderMonkey (Firefox), JSCore (Safari), and Chakra (Edge) work, and it's what gives JavaScript the appearance of being an *interpreted* language.

In this process, steps 1–2 use your program's types; step 3 does not. That's worth reiterating: *when TSC compiles your code from TypeScript to JavaScript, it won't look at your types*. That means your program's types will never affect your program's generated output, and are only used for typechecking. This feature makes it foolproof to play around with, update, and improve your program's types, without risking breaking your application.

The Type System

Modern languages have all sorts of different *type systems*.

TYPE SYSTEM

A set of rules that a typechecker uses to assign types to your program.

There are generally two kinds of type systems: type systems in which you have to tell the compiler what type everything is with explicit syntax, and type systems that infer the types of things for you automatically. Both approaches have trade-offs.[1]

TypeScript is inspired by both kinds of type systems: you can explicitly annotate your types, or you can let TypeScript infer most of them for you.

To explicitly signal to TypeScript what your types are, use annotations. Annotations take the form *value: type* and tell the typechecker, "Hey! You see this *value* here? Its type is *type*." Let's look at a few examples (the comments following each line are the actual types inferred by TypeScript):

```
let a: number = 1                // a is a number
let b: string = 'hello'          // b is a string
let c: boolean[] = [true, false] // c is an array of bo
```

And if you want TypeScript to infer your types for you, just leave them off and let TypeScript get to work:

```
let a = 1                        // a is a number
let b = 'hello'                  // b is a string
let c = [true, false]            // c is an array of bo
```

Right away, you'll notice how good TypeScript is at inferring types for you. If you leave off the annotations, the types are the same! Throughout this book, we will use annotations only when necessary, and let TypeScript work its inference magic for us whenever possible.

NOTE

In general, it is good style to let TypeScript infer as many types as it can for you, keeping explicitly typed code to a minimum.

TypeScript Versus JavaScript

Let's take a deeper look at TypeScript's type system, and how it compares to JavaScript's type system. Table 2-1 presents an overview. A good understanding of the differences is key to building a mental model of how TypeScript works.

Table 2-1. Comparing JavaScript's and TypeScript's type systems

Type system feature	JavaScript	TypeScript
How are types bound?	Dynamically	Statically
Are types automatically converted?	Yes	No (mostly)
When are types checked?	At runtime	At compile time
When are errors surfaced?	At runtime (mostly)	At compile time (mostly)

How are types bound?

Dynamic type binding means that JavaScript needs to actually run your program to know the types of things in it. JavaScript doesn't know your types before running your program.

TypeScript is a *gradually typed* language. That means that TypeScript works best when it knows the types of everything in your program at compile time, but it doesn't have to know every type in order to compile your program. Even in an untyped program TypeScript can infer some types for you and catch some mistakes, but without knowing the types for everything, it will let a lot of mistakes slip through to your users.

This gradual typing is really useful for migrating legacy codebases from untyped JavaScript to typed TypeScript (more on that in "Gradually Migrating from JavaScript to TypeScript"), but unless you're in the middle of migrating your codebase, you should aim for 100% type coverage. That is the approach this book takes, except where explicitly noted.

Are types automatically converted?

JavaScript is weakly typed, meaning if you do something invalid like add a number and an array (like we did in Chapter 1), it will apply a bunch of rules to figure out what you really meant so it can do the best it can with what you gave it. Let's walk through the specific example of how JavaScript evaluates `3 + [1]`:

1. JavaScript notices that `3` is a number and `[1]` is an array.
2. Because we're using `+`, it assumes we want to concatenate the two.
3. It implicitly converts `3` to a string, yielding `"3"`.
4. It implicitly converts `[1]` to a string, yielding `"1"`.
5. It concatenates the results, yielding `"31"`.

We could do this more explicitly too (so JavaScript avoids doing steps 1, 3, and 4):

```
3 + [1];                           // evaluates to "31"

(3).toString() + [1].toString()  // evaluates to "31"
```

While JavaScript tries to be helpful by doing clever type conversions for you, TypeScript complains as soon as you do something invalid. When you run that same JavaScript code through TSC, you'll get an error:

```
3 + [1];                           // Error TS2365: Opera
                                   // types '3' and 'numb

(3).toString() + [1].toString()  // evaluates to "31"
```

If you do something that doesn't seem right, TypeScript complains, and if you're explicit about your intentions, TypeScript gets out of your way. This behavior makes sense: who in their right mind would try to add a number and an array, expecting the result to be a string (of course, besides Bavmorda the JavaScript witch who spends her time coding by candlelight in your startup's basement)?

The kind of implicit conversion that JavaScript does can be a really hard-to-track-down source of errors, and is the bane of many JavaScript programmers. It makes it hard for individual engineers to get their jobs done, and it makes it even harder to scale code across a large team, since every engineer needs to understand the implicit assumptions your code makes.

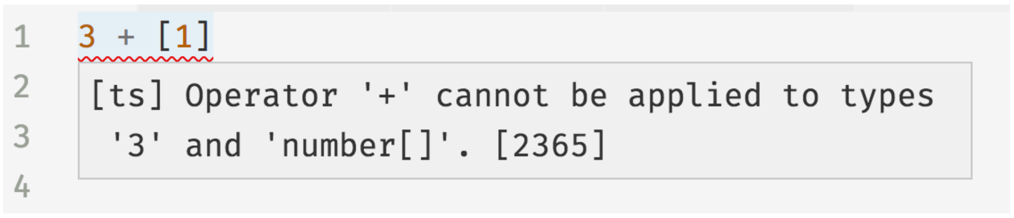
In short, if you must convert types, do it explicitly.

When are types checked?

In most places JavaScript doesn't care what types you give it, and it instead tries to do its best to convert what you gave it to what it expects.

TypeScript, on the other hand, typechecks your code at compile time (remember step 2 in the list at the beginning of this chapter?), so you don't need to actually run your code to see the `Error` from the previous example. TypeScript *statically analyzes* your code for errors like these, and shows them to you before you run it. If your code doesn't compile, that's a really good sign that you made a mistake and you should fix it before you try to run the code.

Figure 2-2 shows what happens when I type the last code example into VSCode (my code editor of choice).



```
3 + [1]
[ts] Operator '+' cannot be applied to types
 '3' and 'number[]'. [2365]
```

Figure 2-2. TypeError reported by VSCode

With a good TypeScript extension for your preferred code editor, the error will show up as a red squiggly line under your code *as you type it*. This dramatically speeds up the feedback loop between writing code, realizing that you made a mistake, and updating the code to fix that mistake.

When are errors surfaced?

When JavaScript throws exceptions or performs implicit type conversions, it does so at runtime.[2] This means you have to actually run your program to get a useful signal back that you did something invalid. In the best case, that means as part of a unit test; in the worst case, it means an angry email from a user.

TypeScript throws both syntax-related errors and type-related errors at compile time. In practice, that means those kinds of errors will show up in your code editor, right as you type—it's an amazing experience if you've never worked with an incrementally compiled statically typed language before.[3]

That said, there are lots of errors that TypeScript can't catch for you at compile time—things like stack overflows, broken network connections, and malformed user inputs—that will still result in runtime exceptions. What TypeScript does is make compile-time errors out of most errors that would have otherwise been runtime errors in a pure JavaScript world.

Code Editor Setup

Now that you have some intuition for how the TypeScript Compiler and type system work, let's get your code editor set up so we can start diving into some real code.

Start by downloading a code editor to write your code in. I like VSCode because it provides a particularly nice TypeScript editing experience, but you can also use Sublime Text, Atom, Vim, WebStorm, or whatever editor you like. Engineers tend to be really picky about IDEs, so I'll leave it to you to decide. If you do want to use VSCode, follow the instructions on the website to get it set up.

TSC is itself a command-line application written in TypeScript,[4] which means you need NodeJS to run it. Follow the instructions on the official NodeJS website to get NodeJS up and running on your machine.

NodeJS comes with NPM, a package manager that you will use to manage your project's dependencies and orchestrate your build. We'll start by using it to install TSC and TSLint (a linter for TypeScript). Start by opening your terminal and creating a new folder, then initializing a new NPM project in it:

```
# Create a new folder
mkdir chapter-2
cd chapter-2

# Initialize a new NPM project (follow the prompts)
npm init

# Install TSC, TSLint, and type declarations for NodeJS
npm install --save-dev typescript tslint @types/node
```

tsconfig.json

Every TypeScript project should include a file called *tsconfig.json* in its root directory. This *tsconfig.json* is where TypeScript projects define things like which files should be compiled, which directory to compile them to, and which version of JavaScript to emit.

Create a new file called *tsconfig.json* in your root folder (`touch tsconfig.json`),[5] then pop it open in your code editor and give it the following contents:

```json
{
  "compilerOptions": {
    "lib": ["es2015"],
    "module": "commonjs",
    "outDir": "dist",
    "sourceMap": true,
    "strict": true,
    "target": "es2015"
  },
  "include": [
    "src"
  ]
}
```

Let’s briefly go over some of those options and what they mean (Table 2-2):

Table 2-2. *tsconfig.json* options

Option	Description
`include`	Which folders should TSC look in to find your TypeScript files?
`lib`	Which APIs should TSC assume exist in the environment you'll be running your code in? This includes things like ES5's `Function.prototype.bind`, ES2015's `Object.assign`, and the DOM's `document.querySelector`.
`module`	Which module system should TSC compile your code to (CommonJS, SystemJS, ES2015, etc.)?
`outDir`	Which folder should TSC put your generated JavaScript code in?
`strict`	Be as strict as possible when checking for invalid code. This option enforces that all of your code is properly typed. We'll be using it for all of the examples in the book, and you should use it for your TypeScript project too.
`target`	Which JavaScript version should TSC compile your code to (ES3, ES5, ES2015, ES2016, etc.)?

These are just a few of the available options—*tsconfig.json* supports dozens of options, and new ones are added all the time. You won't find yourself changing these much in practice, besides dialing in the `module` and `target` settings when switching to a new module bundler, adding `"dom"` to `lib` when writing TypeScript for the browser (you'll learn more about this in Chapter 12), or adjusting your level of `strict` ness when migrating your existing JavaScript code to TypeScript (see "Gradually Migrating from JavaScript to TypeScript"). For a complete and up-to-date list of supported options, head over to the official documentation on the TypeScript website.

Note that while using a *tsconfig.json* file to configure TSC is handy because it lets us check that configuration into source control, you can set most of TSC's options from the command line too. Run `./node_modules/.bin/tsc --help` for a list of available command-line options.

tslint.json

Your project should also have a *tslint.json* file containing your TSLint configuration, codifying whatever stylistic conventions you want for your code (tabs versus spaces, etc.).

NOTE

Using TSLint is optional, but it's strongly recommend for all TypeScript projects to enforce a consistent coding style. Most importantly, it will save you from arguing over code style with coworkers during code reviews.

The following command will generate a *tslint.json* file with a default TSLint configuration:

```
./node_modules/.bin/tslint --init
```

You can then add overrides to this to conform with your own coding style. For example, my *tslint.json* looks like this:

```
{
  "defaultSeverity": "error",
  "extends": [
    "tslint:recommended"
  ],
  "rules": {
    "semicolon": false,
    "trailing-comma": false
  }
}
```

For the full list of available rules, head over to the TSLint documentation. You can also add custom rules, or install extra presets (like for ReactJS).

index.ts

Now that you've set up your *tsconfig.json* and *tslint.json*, create a *src* folder containing your first TypeScript file:

```
mkdir src
touch src/index.ts
```

Your project's folder structure should now look this:

```
chapter-2/
├──node_modules/
├──src/
│ └──index.ts
├──package.json
├──tsconfig.json
└──tslint.json
```

Pop open *src/index.ts* in your code editor, and enter the following TypeScript code:

```
console.log('Hello TypeScript!')
```

Then, compile and run your TypeScript code:

```
# Compile your TypeScript with TSC
./node_modules/.bin/tsc

# Run your code with NodeJS
node ./dist/index.js
```

If you've followed all the steps here, your code should run and you should see a single log in your console:

```
Hello TypeScript!
```

That's it—you just set up and ran your first TypeScript project from scratch. Nice work!

TIP

Since this might have been your first time setting up a TypeScript project from scratch, I wanted to walk through each step so you have a sense for all the moving pieces. There are a couple of shortcuts you can take to do this faster next time:

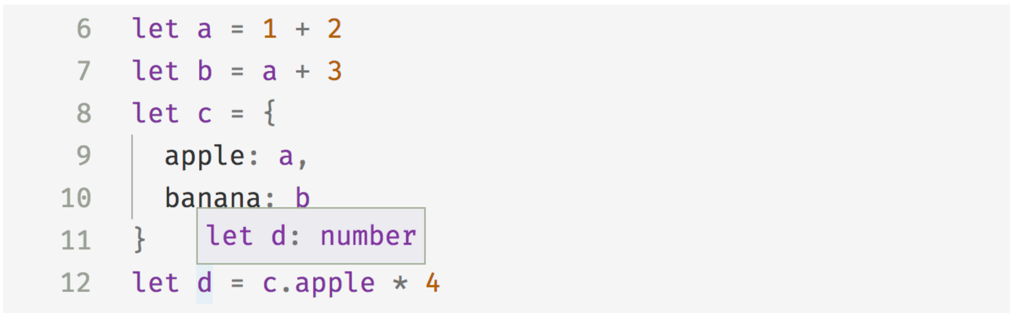
- Install `ts-node`, and use it to compile and run your TypeScript with a single command.
- Use a scaffolding tool like `typescript-node-starter` to quickly generate your folder structure for you.

Exercises

Now that your environment is set up, open up *src/index.ts* in your code editor. Enter the following code:

```
let a = 1 + 2
let b = a + 3
let c = {
  apple: a,
  banana: b
}
let d = c.apple * 4
```

Now hover over `a`, `b`, `c`, and `d`, and notice how TypeScript infers the types of all your variables for you: `a` is a `number`, `b` is a `number`, `c` is an object with a specific shape, and `d` is also a `number` (Figure 2-3).



```
let a = 1 + 2
let b = a + 3
let c = {
  apple: a,
  banana: b
}   let d: number
let d = c.apple * 4
```

Figure 2-3. TypeScript inferring types for you

Play around with your code a bit. See if you can:

- Get TypeScript to show a red squiggly when you do something invalid (we call this “throwing a `TypeError`“).

- Read the `TypeError`, and try to understand what it means.
- Fix the `TypeError` and see the red squiggly disappear.

If you're ambitious, try to write a piece of code that TypeScript is unable to infer the type for.

1 There are languages all over this spectrum: JavaScript, Python, and Ruby infer types at runtime; Haskell and OCaml infer and check missing types at compile time; Scala and TypeScript require some explicit types and infer and check the rest at compile time; and Java and C need explicit annotations for almost everything, which they check at compile time.

2 To be sure, JavaScript surfaces syntax errors and a few select bugs (like multiple `const` declarations with the same name in the same scope) after it parses your program, but before it runs it. If you parse your JavaScript as part of your build process (e.g., with Babel), you can surface these errors at build time.

3 Incrementally compiled languages can be quickly recompiled when you make a small change, rather than having to recompile your whole program (including the parts you didn't touch).

4 This puts TSC in the mystical class of compilers known as *self-hosting compilers*, or compilers that compile themselves.

5 For this exercise, we're creating a *tsconfig.json* manually. When you set up TypeScript projects in the future, you can use TSC's built-in initialize command to generate one for you: `./node_modules/.bin/tsc --init`.