O'REILLY®

Learning TypeScript

Enhance Your Web Development Skills Using Type-Safe JavaScript

Early Release

RAW & UNEDITED

Josh Goldberg

Learning TypeScript

Enhance Your Web Development Skills Using Type-Safe JavaScript

Josh Goldberg

Learning TypeScript

by Josh Goldberg

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Amanda Quinn and Rita Fernando

Production Editor: Beth Kelly

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

September 2022: First Edition

Revision History for the Early Release

- 2021-08-17: First Release

- 2021-11-29: Second Release
- 2022-01-21: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098110338 for release details.

978-1-098-11026-0

[FILL IN]

Dedication

This book is dedicated to my incredible partner Mariah, who introduced me to the joy of adopting backyard cats and has regretted it ever since. Toot.

Preface

My journey to TypeScript was not a direct or quick one. I started off in school primarily writing Java, then C++, and like many new developers raised on statically typed languages, I looked down on JavaScript as “just” the sloppy little scripting language people throw onto websites.

My first substantial project in the language was a silly remake of the original *Super Mario Bros.* video game in pure HTML5/CSS/JavaScript and, typical of many first projects, was an absolute mess. In the beginning of the project I instinctively disliked JavaScript’s weird flexibility and lack of guardrails. It was only towards the end that I really began to respect JavaScript’s features and quirks: its flexibility as a language, its ability to mix and match small functions, and its ability to *just work* in user browsers within seconds of page load.

By the time I finished that first project, I had fallen in love with JavaScript.

Static analysis (tools to analyze your code without running it) such as TypeScript also gave me a queasy gut feeling at first. *JavaScript is so breezy and fluid*, I thought, *why bog ourselves down with rigid structures and types?*. Were we reverting back to the worlds of Java and C++ that I had left behind?

It took me all of ten minutes of struggling to read through my old, convoluted JavaScript code to understand how messy things could get without static analysis. The act of cleaning that code up showed me all the places where I would have benefitted from some structure. From that point on, I was hooked onto adding as much static analysis to my projects as I could.

It’s been nearly a decade since I first tinkered with TypeScript and I enjoy it as much as ever. The language is still evolving with new features and more useful than ever in providing *safety* and *structure* to JavaScript.

I hope that by reading *Learning TypeScript* you can learn to appreciate TypeScript the way I do: not just as a means to find bugs and typos --and certainly not a substantial change to JavaScript code patterns-- but as JavaScript *with types*: a beautiful system for declaring the way our JavaScript *should* work, and helping us stick to it.

Who Should Read This Book

If you have an understanding of writing JavaScript code, can run basic commands in a terminal, and are interested in learning about TypeScript, this book is for you.

Maybe you've heard TypeScript can help you write a lot of JavaScript with fewer bugs *(true!)* or document your code well for other people to read *(also true!)*. Maybe you've seen TypeScript show up in a lot of job postings, or in a new role you're starting.

Whatever your reason, as long as you come in knowing the fundamentals of JavaScript -variables, functions, closures/scope, and classes- this book will take you from no TypeScript knowledge to mastering the fundamentals and most important features of the language. By the end of this book you will understand:

- The history and context for why TypeScript is useful on top of "vanilla' JavaScript
- How a type system analyzes and understands code
- How to use development-only type annotations to inform the type system
- How TypeScript works with IDEs to provide refactoring tools

And you will be able to:

- Articulate the benefits of TypeScript and general characteristics of its type system

- Add type annotations where useful in your code
- Represent moderately complex types using TypeScript's built-in inferences and new syntax
- Use TypeScript to assist local development in refactoring code

Why I Wrote This Book

TypeScript is a wildly popular language in both industry and open source:

- GitHub's 2020 State of the Octoverse has it at the 4th top language, up from 7th in 2019 and 2018 and 10th in 2017.
- StackOverflow's 2020 Developer Survey has it at the world's second most loved language (67.1% of users).
- The 2020 State of JS survey has TypeScript at consistently having both high satisfaction and usage amounts as both a build tool and variant of JavaScript.

For frontend developers, TypeScript is well supported in all major UI libraries and frameworks, including Angular -which strongly recommends TypeScript-, Gatsby, Next.js, React, Svelte, and Vue. For backend developers, TypeScript generates JavaScript that runs natively in Node.js; Deno, a similar runtime by Node's creator, emphasizes directly supporting TypeScript files.

However, despite this plethora of popular project support, I was rather disappointed by the lack of good introductory content online when I first learned the language. Many of the online documentation sources didn't do a great job of explaining what a "type system" is or how to use it. They often assumed a great deal of prior knowledge in both JavaScript and strongly typed languages, or were written with only cursory code examples.

Not seeing an *O'Reilly* book with a cute animal cover introducing TypeScript was a disappointment. I'm thrilled to be able to spend the time to really make a clear, comprehensive introduction to TypeScript for readers who aren’t already experts in its principles.

Navigating This Book

Learning TypeScript has two purposes:

1. You can read through it once to to understand TypeScript as a whole.
2. Later, you can refer back to it as a practical introductory TypeScript language reference.

This book ramps up from concepts to practical use across three general sections of chapters:

- **Part 1: Concepts**: How JavaScript came to be, why TypeScript adds to it, and the basics of a *type system* as TypeScript creates it.
- **Part 2: Features**: Fleshing out how the type system interacts with the major parts of JavaScript you’d work with when writing TypeScript code.
- **Part 3: Usage**: Now that you understand the features that make up the TypeScript language, how to use them in real world situations to improve your code reading and editing experience.

I’ve thrown in one last chapter after that on “Cursory Miscellaneous Topics” to cover fancy TypeScript features. You won’t need to know them to consider yourself a TypeScript developer (and arguably most should be avoided whenever possible). They’re just nifty and interesting enough that I couldn’t resist giving them a mention.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/LearningTypeScript/learning-typescript*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

O'Reilly Online Learning

training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Part I. Concepts

Chapter 1. From JavaScript to TypeScript

JavaScript (ECMAScript)

Supports browsers decades past

Beauty of the web

Before talking about TypeScript, we need to first understand where it came from: JavaScript!

History of JavaScript

JavaScript was originally designed in 10 days by Brendan Eich at Netscape in 1995 to be approachable and easy to use for websites. Developers have been poking fun at its quirks and perceived shortcomings ever since. I'll cover some of them in the next section.

JavaScript has evolved tremendously since 1995, though! Its steering committee, TC39, has released new versions of the language consistently yearly since 2015 with new features that bring it in line with other modern dynamic languages. Impressively, even with regular new language versions, JavaScript has managed to maintain backwards compatibility for decades in varying environments, including browsers, embedded applications, and server runtimes.

Today, JavaScript is a wonderfully flexible language with a lot of strengths. You and I should appreciate that while JavaScript has its quirks, it's also helped enable the incredible growth of web pages and the internet.

> *Show me the perfect programming language and I'll show you a language with no users.*
>
> —Anders Hejlsberg, TSConf 2019

Vanilla JavaScript's Pitfalls

Developers often refer to using JavaScript without any significant language extensions or frameworks as "vanilla": referring to it being the familiar, original plain version. I'll soon go over why TypeScript adds just the right flavor to overcome these particular major pitfalls, but it's useful to understand just why they can be painful.

Costly Freedom

Many developers' biggest gripe with JavaScript is unfortunately one of its key feature: JavaScript provides virtually no restrictions in how you structure your code. That freedom makes it a ton of fun to start a project in JavaScript! You can quickly write code in any which way and watch it magically work.

As you get to have more and more files, though, it becomes apparent how that freedom can be damaging. Take the following snippet, presented out of context from some fictional musical application:

```
function paintPainting(painter, painting) {
  return painter
    .prepare()
    .paint(painting, painter.ownMaterials)
    .finish();
}
```

Reading that code without any context, you can only have vague ideas how to use `paintPainting`. Perhaps if you've worked in the surrounding codebase you may recall that `painter` should be what's returned by some `getPainter` function. You might even make a lucky guess that `painting` is a `string`.

Even if those assumptions are correct, though, later changes to the code may invalidate them. Perhaps `painting` is changed from a `string` to some other data type, or maybe one or more of the painter's methods are renamed.

Other languages might refuse to let you run code if their compiler determines it would likely crash. Not so with JavaScript.

The freedom of code that makes JavaScript so fun becomes a real pain when you want safety in running your code.

Loose Documentation

Nothing exists in the JavaScript language specification to formalize describing what function parameters, function returns, variables, or other constructs in code are meant to be. Many developers have adopted a standard called JSDoc to describe functions and variables using block comments. The JSDoc standard describes how you might write documentation comments placed directly above constructs such as functions and variables, formatted in standard way. Here's an example, again taken out of context:

```
/**
 * Performs a painter painting a particular painting.
 *
 * @param {Hero} painter
 * @param {string} painting
 * @returns {boolean} Whether the painter painted the painting.
 */
function paintPainting(painter, painting) { /* ... */ }
```

JSDoc has key issues that often make it unpleasant to use in a large codebase:

- It can be difficult during code refactors to find all the now-invalid JSDoc comments related to your changes
- Nothing stops JSDoc descriptions from being wrong about code
- Describing complex objects is unwieldy, requiring multiple standalone comments to define types and their relationships

Weaker Developer Tooling

Because JavaScript doesn't provide built-in ways to identify types, and code easily diverge from JSDoc comments, it can be very difficult to automate large changes to or gain insights about a codebase. JavaScript developers are often surprised to see features in typed languages such as C# and Java that allow developers to perform class member renamings or jump to the place an argument's type was declared.

NOTE

You may protest that modern IDEs such as VS Code do provide some automated refactors to JavaScript. True, but: they use TypeScript to do so, and they're not as reliable or as powerful as when refactoring well-defined TypeScript code.

TypeScript!

TypeScript was created internally at Microsoft in the early 2010s then released and open sourced in 2012. The head of its development is Anders Hejlsberg, notable for also having lead the development of the popular C#

and Turbo Pascal languages. TypeScript is often described as a *“superset of JavaScript”* or *“JavaScript with types”*.

But what *is* TypeScript?

TypeScript is four things:

- **Programming language**: a language that includes all the existing JavaScript syntax, plus new TypeScript-specific syntax for defining and using types.
- **Type checker**: a program that takes in a set of files written in JavaScript and/or TypeScript, develops an understanding of all the constructs (variables, functions, …) created, and lets you know if it thinks anything is set up incorrectly.
- **Compiler**: a program that runs the type checker, reports any issues, then outputs the equivalent JavaScript code.
- **Language services**: a program that uses the type checker to tell editors such as VS Code how to provide helpful utilities to developers.

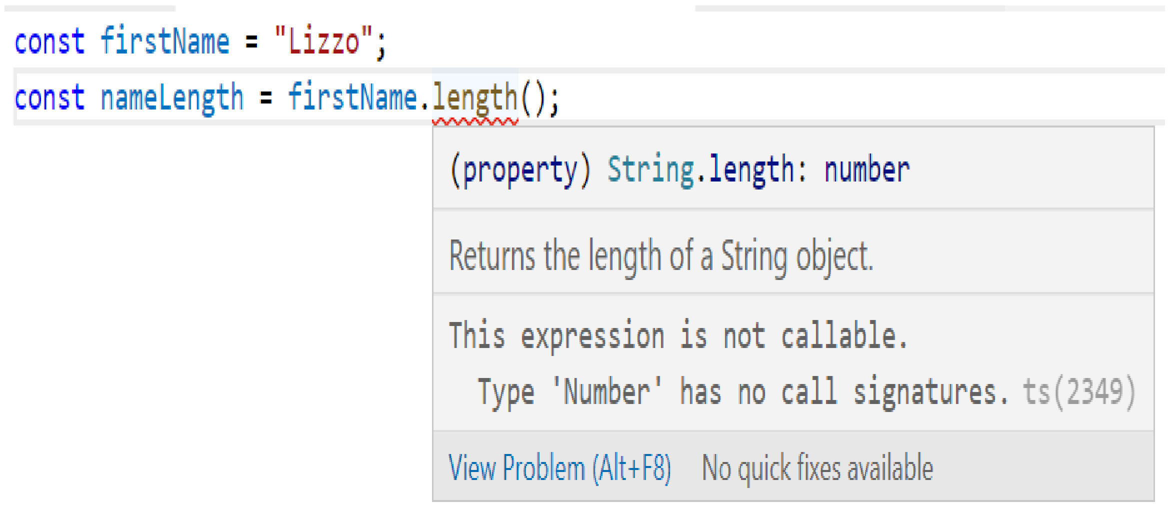
TypeScript in Action

Take a look at this code snippet:

```
const firstName = "Georgia";
const nameLength = firstName.length();
//                           ~~~~~~
// This expression is not callable.
```

The code is written in normal JavaScript syntax — I haven’t introduced TypeScript-specific syntax yet. If you were to run the TypeScript type checker on this code, it would use its knowledge that the `length` property of a string is a number -not a function- to give you the complaint shown in the comment.

If you were to paste that code into an editor with TypeScript support - meaning it runs the TypeScript language services for you-, it would be told by those language services to give you a little red squiggly under `length` indicating TypeScript's displeasure with your code. Hovering over the squigglied code would give you the text of the complaint (Figure 1-1):



```
const firstName = "Lizzo";
const nameLength = firstName.length();
```

```
(property) String.length: number

Returns the length of a String object.

This expression is not callable.
  Type 'Number' has no call signatures. ts(2349)
```

Figure 1-1. TypeScript reporting an error on string length not being callable.

Being told of these simple errors in your editor *as you type them* is a lot more pleasant than waiting until a particular line of code happens to have been run in the app and throw an error.

Freedom Through Restriction

TypeScript allows us to specify what types may be provided as values for parameters and variables. Some developers find having to explicitly write out in your code how particular areas are supposed to work to be restrictive at first.

But! I would argue that being "restricted" in this way is actually a good thing! By restricting our code to only being able to be used in the ways you specify, TypeScript can give you confidence that changes in one area of code won't break other areas of code that use it.

If, say, you change the number of required parameters for a function, TypeScript will let you know if you forget to update a place that calls the function.

In the following example, `sayMyName` was changed from taking in two parameters to taking one parameter, but the call to it with two strings wasn't updated and so is triggering a TypeScript complaint:

```
// Previously: sayMyName(firstName, lastNameName) { ...
function sayMyName(fullName) {
  console.log(`You acting kind of shady, ain't callin' me
${fullName}`);
}

sayMyName("Beyoncé", "Knowles");
//                   ~~~~~~~~~
// Expected 1 arguments, but got 2.
```

Precise Documentation

Let's look at a TypeScript-ified version of the `paintPainting` function from earlier. Although I haven't yet gone over the specifics of TypeScript syntax for documenting types, the following snippet still hints at the great precision with which TypeScript can document code:

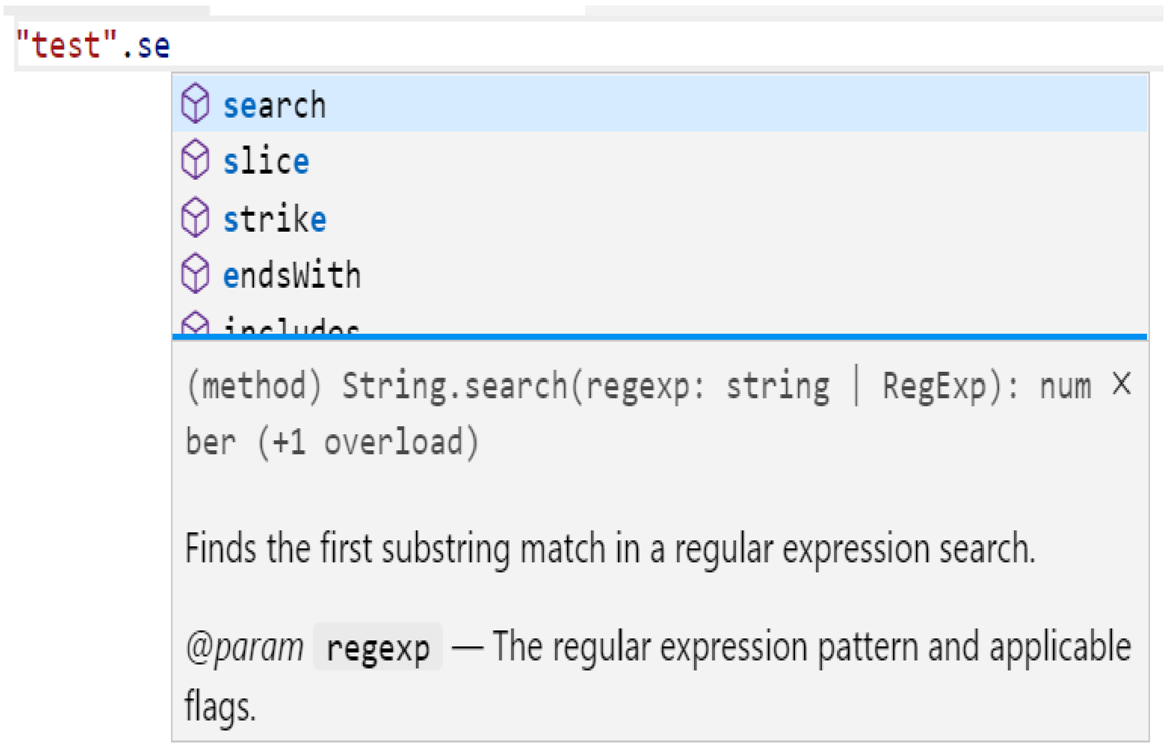
```
interface Painter {
  finish(): boolean;
  ownMaterials: Materials;
  paint(painting: string, materials: Material[]);
}

function paintPainting(painter: Painter, painting: string) { /*
... */ }
```

A TypeScript developer reading this code for the first time could understand that `painter` has at least three properties, two of which are methods. By baking in syntax to describe of the "shapes" of objects, TypeScript provides an excellent, *precise* method for describing how objects look.

Stronger Developer Tooling

TypeScript's precise typings allow editors such as VS Code to gain a much deeper insight into your code, which they then use to surface intelligent suggestions as you type. These suggestions can be incredibly useful for development.

To start, if you've used VS Code to write JavaScript before, you might have noticed that it suggests "autocompletions" as you write code with built-in types of objects like strings (Figure 1-2):



```
"test".se
```

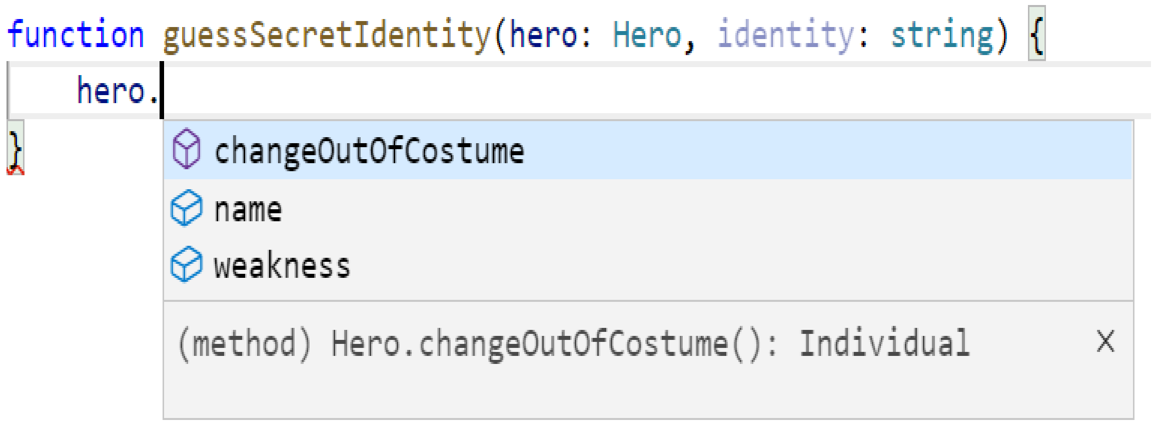
```
(method) String.search(regexp: string | RegExp): number (+1 overload)
```

Finds the first substring match in a regular expression search.

@param `regexp` — The regular expression pattern and applicable flags.

Figure 1-2. TypeScript providing autocompletion suggestions in JavaScript for a string.

When you add TypeScript's type checker for understanding code, it can give you these rich suggestions even for code you've written. If start typing `painter.` in the `paintPainting` function, TypeScript would take its knowledge that the `painter` parameter is of type `Painter` and the `Painter` type has the following members (Figure 1-3):



```
function guessSecretIdentity(hero: Hero, identity: string) {
    hero.
}
```

```
(method) Hero.changeOutOfCostume(): Individual
```

Figure 1-3. TypeScript providing autocompletion suggestions in TypeScript for a custom type.

Snazzy!

What TypeScript Is Not

Now that you've seen how awesome TypeScript is, before you give it a test drive, I have to warn you about some limitations. It's still a very fantastic piece of technology but it's important to check expectations regarding what TypeScript is, and isn't. As with every tool, it excels at some areas and has limitations in others.

A Remedy for Bad Code

TypeScript helps you structure your JavaScript but other than enforcing type safety it doesn't enforce any opinions on what that structure should look like.

Good!

TypeScript is a language that everyone is meant to be able to use, not an opinionated framework with a target audience.

If anybody tries to tell you that TypeScript forces you to use classes, or makes it hard to write good code, or whatever code style complaints are out there, give them a stern look and tell them to pick up a copy of *Learning TypeScript*. TypeScript does not enforce code style opinions such as whether to use classes or functions.

Extensions to JavaScript (Mostly)

TypeScript's design goals explicitly state that it should:

- Align with current and future ECMAScript proposals.
- Preserve runtime behavior of all JavaScript code.

TypeScript does not change how JavaScript works at all. Its creators have tried very hard to avoid adding new code features that would add to or conflict with JavaScript. Such a task is the domain of TC39, the technical committee that works on ECMAScript itself.

There are a few older features in TypeScript that were added many years ago to reflect common use cases in JavaScript code. Most of those features are either relatively uncommon or have fallen out of favor, and are only covered briefly in *Appendix ??*. I recommend staying away from them in most cases.

Slower Than JavaScript

Sometimes on the internet, you might hear some opinionated developers complain that TypeScript is slower than JavaScript at runtime. That claim is generally misleading and inaccurate: the only changes TypeScript makes to code are if you ask it to transpile your code down to earlier versions of JavaScript to support older runtime environments such as JavaScript. Many production frameworks don't use TypeScript's transpiler at all, instead using a separate tool for transpilation and TypeScript only for type checking.

TypeScript does, however, add some time to building your code. TypeScript code must be compiled down to JavaScript before most environments, such as browsers and Node.js, will run it. Most build pipelines are generally set up so that the performance hit is negligible, and expensive TypeScript-only operations are done separately from truly building code to be run.

WARNING

Even projects that seemingly allow running TypeScript code directly, such as `ts-node` and Deno, themselves internally convert TypeScript code to JavaScript before running it.

Finished Evolving

The web is nowhere near finished evolving, and thus neither is TypeScript. The TypeScript language is constantly receiving bug fixes and feature additions to match the ever-shifting needs of the web community. The basic tenets of TypeScript you'll learn in this book will remain about the same, but error messages, fancier features, and editor integrations will improve over time.

In fact, while this edition of the book was published with TypeScript version 4.3.5 as the latest, by the time you started reading it we can be certain a newer version has been released. Some of the TypeScript error messages in this book might even already be out of date!

Getting Started in the TypeScript Playground

You've read a good amount about TypeScript by now. Let's get you writing it!

The main TypeScript website includes a "playground" editor at *https://www.typescriptlang.org/play*. You can type code into the main editor and see many of the same editor suggestions you would working with TypeScript locally in a= full IDE.

Most of the snippets in this book are intentionally small and self-contained enough that you could type them out in the playground and tinker with them for fun.

Compiling Syntax

TypeScript's compiler allows us to input TypeScript syntax, have it type checked, and get the equivalent JavaScript emitted. As a convenience, the compiler may also take modern JavaScript syntax and compile it down into its older ECMAScript equivalents.

If you were to paste this TypeScript code into the playground:

```
const artist = "Augusta Savage";
console.log({ artist });
```

The playground would show you on the right-hand-side of the screen that this would be the equivalent JavaScript output by the compiler:

```
"use strict";
var artist = "Augusta Savage";
console.log({ artist: artist });
```

The TypeScript playground is a great tool for showing how source TypeScript becomes output JavaScript.

NOTE

Many JavaScript projects instead use dedicated transpilers such as Babel (*https://babeljs.io*) to transpile source code into runnable JavaScript. Babel does have great TypeScript tooling, though I don't cover it until *Chapter ??*.

Getting Started Locally

To install the latest version of TypeScript globally on your computer, run the following command:

```
npm i -g typescript
```

Now, you'll be able to run TypeScript on the command-line with the `tsc` (**T**ype**S**cript **C**ompiler) command. Try it with the `--version` command to make sure it's set up properly.

```
tsc --version
```

It should print out something like `Version X.Y.Z` — whichever version is current as of you installing TypeScript.

Running Locally

Now that TypeScript is installed, let's have you set up a folder locally to run TypeScript on code. Create a folder somewhere on your computer - for my commands, I'll assume the path `/code/learning-typescript` - and run this command to create a new `tsconfig.json` configuration file:

```
tsc --init
```

A `tsconfig.json` file declares the settings that TypeScript uses when analyzing your code. Most of the options in that file aren't going to be relevant to you in this book (there are a lot of uncommon edge cases in programming that the languages need to account for!). I'll cover them in *Chapter ??*.

The important feature is that now you can run `tsc` to tell TypeScript to compile all the files in that folder.

Try adding a file named `index.ts` with the following contents:

```
console.blub("Nothing is worth more than laughter.");
```

Then, run `tsc` and provide it the name of that `index.ts` file:

```
tsc index.ts
```

You should get an error that looks roughly like:

```
index.ts:1:9 - error TS2339: Property 'blub' does not exist on
type 'Console'.

1 console.blub("Nothing is worth more than laughter.");
          ~~~~
```

```
Found 1 error.
```

Indeed, `blub` does not exist on the `console`. What was I thinking?

Before you fix the code to appease TypeScript, note that `tsc` created an `index.js` for you with contents including the `console.blub`.

> **NOTE**
>
> This is an important concept: even though there was a *type error* in our code, the *syntax* was still completely valid. The TypeScript compiler will still attempt to produce JavaScript from an input file's syntax regardless of any type errors.

Correct the code in `index.ts` to call `console.log` and run `tsc` again. There should be no complaints in your terminal, and the `index.js` file should now contain updated output code.

```
"use strict";
console.log("Nothing is worth more than laughter.");
```

Editor Features

Another benefit of creating a `tsconfig.json` file is that when editors are opened to a particular folder, they will now recognize that folder as a TypeScript project. For example, if you open VS Code in the same folder through its GUI or with its terminal command:

```
code .
```

(alternately, `code /code/learning-typescript`)

The settings it uses to analyze your TypeScript code will respect whatever's in the `tsconfig.json` file it sees in that folder.

As an exercise, go back through the code snippets in this chapter and type them in your editor. You should see dropdowns suggesting completions for names as you type them, especially for members such as the `log` on `console`.

Very exciting: you’re using the TypeScript language services to help yourself write code! You’re on your way to being a TypeScript developer!

TIP

VS Code comes with great TypeScript support. Many TypeScript and/or web developers use VS Code as their primary editor — myself included. You don’t *have* to use it for TypeScript, but I do recommend it for at least trying out TypeScript while reading through this book.

Summary

In this chapter, you read up on the context for some of JavaScript’s main weaknesses and how TypeScript helps with:

- The history of JavaScript
- JavaScript’s pitfalls: costly freedom, loose documentation, and and weaker developer tooling
- TypeScript’s advantages: freedom through restriction, precise documentation, and stronger developer tooling
- What TypeScript is not: a remedy for bad code, extensions to JavaScript (mostly), slower than JavaScript, or finished evolving
- Getting started writing TypeScript code on the TypeScript playground and locally on your computer

Chapter 2. The Type System

JavaScript's power

Comes from flexibility

Be careful with that!

I talked briefly in Chapter 1 about the existence of a "type system" in TypeScript, where its type checker looks at your code, understands how it's meant to work, and lets you know where you might have messed up. But how does a type system work, really?

What's in a Type?

A "type" is a description of what a JavaScript value *shape* might be. By *shape* I mean how a value looks and feels: its properties and methods, and what the built-in `typeof` operator would describe it as. The most basic types in TypeScript correspond to the seven basic primitives in JavaScript:

- **`bigint`**: `0n`, `2n`, `-4n`, …
- **`boolean`**: `true` or `false`
- **`null`**
- **`number`**: `0`, `2`, `-4`, …
- **`string`**: `""`, `"Hi!"`, `"abc123"`, …
- **`symbol`**: `Symbol()`, `Symbol("hi")`, …
- **`undefined`**

For example, when you create a variable with the initial value `"Aretha"`:

```
let singer = "Aretha";
```

TypeScript can figure out, or infer, that the `singer` variable is of *type* `string`.

For each of these values, TypeScript understands the type of the value to be one of the seven basic primitives:

```
1337n; // bigint
true; // boolean
null; // null
1337; // number
"Louise"; // string
Symbol("Franklin"); // Symbol
undefined; // undefined
```

TypeScript is also smart enough to figure out the type of a variable whose starting value is computed. In this example, TypeScript knows that the ternary expression always results in a string, so the `bestSong` value is a `string`:

```
let bestSong = Math.random() > 0.5 ? "Chain of Fools" :
"Respect";
```

NOTE

Recall the differences between objects and primitives in JavaScript: classes such as `Boolean` and `Number` wrap around their primitive equivalents. TypeScript best practice is generally to refer to the lower-case names, such as `boolean` and `number`, respectively.

Type Inferences in Detail

At its core, TypeScript's type system works by:

1. Reading in your code and understanding all the types and values in existence
2. For each object, seeing what type its initial declaration indicates it may contain
3. For each object, seeing all ways it's used later on
4. Complaining to the user if an object's usage doesn't match with its type

Let's walk through that type inference system in detail.

Take the following snippet, in which TypeScript is emitting a type error about a member variable being erroneously called as a function:

```
let firstName = "Whitney";
firstName.length();
//        ~~~~~~
//  This expression is not callable.
//    Type 'Number' has no call signatures
```

TypeScript came to that complaint by, in order:

1. Reading in the code and understanding there to be one object: `firstName`
2. Concluding that `firstName` is of type `string` because its initial value is a `string`, `"Cleopatra"`

3. Seeing that the code is trying to access a `.length` member of `firstName` and call it like a function
4. Complaining that the `.length` member of a string is a number, not a function *(it can't be called like a function)*

Understanding TypeScript's type inference is an important skill for understanding TypeScript code. Code snippets in this chapter and through the rest of this book will display more and more complex types that TypeScript will be able to infer from code.

Kinds of Errors

While writing TypeScript, the two kinds of "errors" you'll come across most frequently are:

- **Syntax**: blocking TypeScript from being converted to JavaScript.
- **Type**: something mismatched has been detected by the type checker.

It's useful to be able to understand the differences between the two.

Syntax Errors

Syntax errors are when TypeScript detects incorrect syntax that it cannot understand as code. These block TypeScript from being able to properly generate output JavaScript from your file. Depending on the tooling and settings you're using to convert your TypeScript code to JavaScript, you might to still get *some* kind of JavaScript output. If you do, it likely won't look like what you expect.

This input TypeScript has a syntax error for an unexpected `let`:

```
let let wat;
//      ~~~
// Error: ',' expected.
```

Its compiled output in TypeScript, depending on the language version, may look something like:

```
let let, wat;
```

Although TypeScript will do its best to output JavaScript code regardless of syntax errors, the output code will likely not be what you wanted. It's best to fix syntax errors before attempting to run the output JavaScript.

Type Errors

Type errors occur when your syntax is valid but the TypeScript type checker has detected an error with the program's types. These do not block TypeScript syntax from being converted to JavaScript. They do, however, often indicate something will crash or behave unexpectedly if your code is allowed to run.

You saw this in the Chapter 1 with the `console.blub` example, where it is syntactically valid code but TypeScript can detect it will likely crash when run:

```
console.blub("Nothing is worth more than laughter.");
//      ~~~~
// Error: Property 'blub' does not exist on type 'Console'.
```

Even though TypeScript will output valid JavaScript code, the type errors are generally a sign that the output JavaScript likely won't run the way you wanted. It's best to read them and consider fixing any reported issues before running JavaScript.

NOTE

Some projects are configured to block running code during development until all TypeScript type errors -not just syntax- are fixed. Many developers, myself included, generally find this to be annoying and unnecessary. Most projects have a way to disable it, such as with the `tsconfig.json` file covered in *Chapter ??*.

Assignability

TypeScript reads variables’ initial values to determine what type those variables are allowed to be. If it later sees an assignment of a new value to that variable, it will check if that assignment value’s type is the same as the variable’s.

TypeScript would be fine with later assigning a different value of the same type to a variable. If a variable is, say, initially a *string* value, later assigning it another *string* would be fine:

```
let firstName = "Carole";
firstName = "Louise";
```

When a variable is declared with an initial value, it’s considered to be of that initial value’s type — and must always be assigned values of that type. If TypeScript sees an assignment of a different type, it will give us type error. We couldn’t, say, initially declare a variable with a *string* value and then later on put in a *number*:

```
let firstName = "King";
firstName = 1337;
// Error: Type 'number' is not assignable to type 'string'.
```

TypeScript’s checking of whether a value is allowed to be provided to a function call or variable is called “assignability”: whether that value is *assignable* to the location it’s passed to. This will be an important term in later chapters as we compare more complex objects.

Type Annotations

Sometimes a variable doesn’t have an initial value for TypeScript to read. TypeScript won’t attempt to figure out what value those variables are from later usage. It’ll consider them to be implicitly the `any` type: a type indicating that it could be *anything* in the world.

```
let rocker;
rocker = "Joan Jett";
```

Allowing variables to be of type `any` defeats the purpose of TypeScript's type checking! TypeScript works best when it knows what types your values are meant to be. Most of TypeScript's type checking can't be applied to `any` typed values because they don't have known types to be checked.

Although TypeScript can still emit JavaScript code despite an implicit `any`, it will yell at you about them in the form of type errors. *Chapter ??* will cover how to configure TypeScript's implicit `any` complaints.

Instead, TypeScript provides a syntax for declaring the type of a variable, using what's called a *type annotation*. A type annotation is placed after the name of a variable and includes a colon followed by the name of a type.

```
let rocker: string;
rocker = "Joan Jett";
```

These type annotations are unique to TypeScript. If you run `tsc` to compile TypeScript source code to JavaScript, they'll be erased. For example:

```
// output .js file
let rocker;
rocker = "Joan Jett";
```

You'll see through the next few chapters how type annotations allow you to augment TypeScript's insights into your code, allowing it to give you better features during development.

NOTE

TypeScript contains an assortment of new pieces of syntax such as these type annotations that exist only in the type system. Nothing that exists only in the type system gets copied over into emitted JavaScript.

Unnecessary Type Annotations

Type annotations allow us to provide information to TypeScript that it wouldn't have been able to glean on its own. You could use them on variables that have immediately inferable types, but you wouldn't be telling TypeScript anything it doesn't already know:

```
let firstName: string = "Tina";
//             ~~~~~~~~ Does not change the type system...
```

If you do add a type annotation to a variable with an initial value, TypeScript will check that it matches the type of the variable's value.

The following `firstName` is declared to be of type `string` but its initializer is the `number` `42`, which is TypeScript sees as an incompatibility:

```
let firstName: string = 42;
//  ~~~~~~~~~
// Error: Type 'number' is not assignable to type 'string'.
```

NOTE

Many developers -myself included- generally prefer not to add type annotations in places where they don't change anything. Having to manually write out type annotations can be cumbersome — especially when they change, and for the complex types I'll show you later in this book.

Type Shapes

TypeScript doesn't only check that the values assigned to variables match their original types: it also knows what member properties should exist on objects. If you attempt to access a property of a variable, TypeScript will make sure that property is known to exist on that variable's type.

Suppose we declare a `rapper` variable of type `string`. Later on, when we use that `string` variable, operations that TypeScript knows work on strings are allowed:

```
let rapper = "Queen Latifah";
rapper.length; // ok
```

Operations that TypeScript doesn’t know to work on strings will not be allowed:

```
rapper.asdfqwerty;
//     ~~~~~~~~~~
// Property 'asdfqwerty' does not exist on type 'string'.
```

Types can also be more advanced shapes, most notably complex objects. In the following snippet, TypeScript knows the `birthNames` object doesn’t have a `rihanna` key and complains:

```
let cher = {
  firstName: "Cherilyn",
  lastName: "Sarkisian",
};

cher.middleName;
//   ~~~~~~~~~~
//   Property 'middleName' does not exist on type
//   '{ firstName: string; lastName: string; }'.
```

TypeScript’s understanding of object shapes allows it to report issues with the usage of objects, not just assignability. Chapter 7 will describe more of TypeScript’s powerful features around objects and object types.

Summary

In this chapter, you saw how TypeScript’s type system works at its core:

- What a “type” is and what types are recognized by TypeScript
- Inferred variable types and variable assignability
- Type annotations to explicitly declare variable types
- Object member checking on type shapes

- How type complaints compare to syntax complaints

Chapter 3. Unions and Narrowing

Nothing is constant

Values may change over time

(well, except constants)

Chapter 2 covered the concept of the "type system" and how it can read values to understand the types of variables. Before I talk about all the various syntax features of TypeScript, I'd like to introduce two key concepts that TypeScript works with to make inferences on top of those values:

- **Unions**: Expanding a value's allowed type to be two or more possible types.
- **Narrowing**: Reducing a value's allowed type to *not* be one or more possible types.

Put together, unions and narrowing are a powerful concept that allow TypeScript to make informed inferences on your code many other mainstream languages cannot.

Union Types

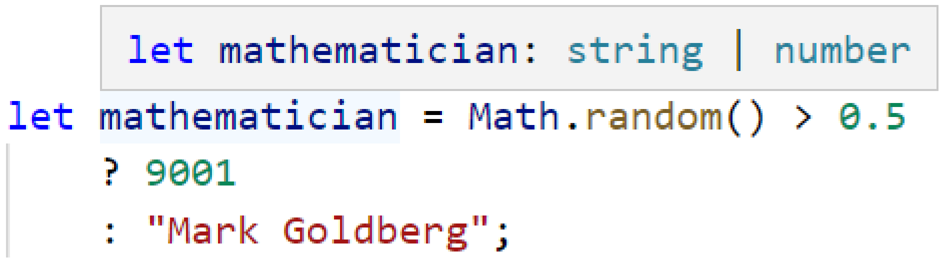
Take this `mathematician` variable:

```
let mathematician = Math.random() > 0.5
    ? 9001
    : "Mark Goldberg";
```

What type is `mathematician`?

It's neither only `number` nor only `string`, even though those are both potential values. `mathematician` can be *either* `number` or `string`. This kind of "either or" type is called a "union". Union types are a wonderful concept that let us handle code cases where we don't know exactly which type a variable is, but do know it's one of a two or more options.

TypeScript represents union types to us using the `|` (pipe) operator between the possible values. The above `mathematician` type is thought of as `number | string`.



```
let mathematician: string | number
```

```
let mathematician = Math.random() > 0.5
    ? 9001
    : "Mark Goldberg";
```

Figure 3-1. TypeScript reporting the mathematician variable as being type string | number.

Declaring Union Types

Union types are an example of a time when it might be useful to give an explicit type annotation for a variable even though it has an initial value. In this example, `thinker` starts off `false` but is known to potentially contain a `string` instead:

```
let thinker: boolean | string = false;

if (Math.random() > 0.5) {
    thinker = "Susanne Langer";
}
```

Union type declarations can be placed anywhere you might declare a type with a type annotation.

NOTE

The order of a union type declaration does not matter. You can write `boolean | number` or `number | boolean` and TypeScript will treat both the exact same.

Union Properties

When a value is known to be a union type, TypeScript will only allow you to access member properties that exist on *all* possible types in the union. It will give you a type checking error if you try to access a type that doesn't exist on all possible types.

In the following snippet, `physicist` is of type `number | string`, so while `.toString()` exists in both types and is allowed to be used, `.toUpperCase()` and `.toFixed()` are not because `.toUpperCase()` is missing on the `number` type and `.toFixed()` is missing on the `string` type:

```
let physicist = Math.random() > 0.5
    ? "Marie Curie"
    : 84;

physicist.toString(); // Ok
physicist.toUpperCase();
//        ~~~~~~~~~~~
// Error: Property 'toUpperCase' does not exist on type 'string |
number'.
//   Property 'toUpperCase' does not exist on type 'number'.

physicist.toFixed();
//        ~~~~~~~
// Error: Property 'toFixed' does not exist on type 'string |
number'.
//   Property 'toFixed' does not exist on type 'string'.
```

Restricting access to properties that don't exist on all union types is a safety measure. If an object is not known to definitely be a type that contains a member, TypeScript will believe it unsafe to try to use that member. It might not exist!

To use a member of a union typed value that only exists on a subset of the potential types, your code will need to indicate to TypeScript that the value is one of those more specific types: a process called *narrowing*.

Narrowing

Narrowing is when TypeScript infers from your code that a value is of a more specific type than what it was defined, declared, or previously inferred as. Once TypeScript knows that a type is more narrow than previously known, it will allow you to treat it like that more specific type.

I'll use the remainder of this chapter to show you common ways TypeScript can deduce type narrowing from your code.

Assignment Narrowing

If you directly assign a value to a variable, TypeScript will understand the variable to have been narrowed to that value's type.

Here, the `admiral` variable is declared initially as a `number | string`, but after it is assigned the value `"Grace Hopper"` TypeScript knows it must be a `string`:

```
let admiral: number | string;

admiral = "Grace Hopper";

admiral.toUpperCase(); // Ok: string

admiral.toFixed();
//         ~~~~~~~
// Error: Property 'toFixed' does not exist on type 'string'.
```

Assignment narrowing comes into play when a variable is given an explicit union type annotation and an initial value, too. TypeScript will understand that while the variable may later receive a value of any of the union typed values, it starts off as only the type of its initial value.

In the following snippet, `inventor` is declared as type `number | string`, but TypeScript knows it's immediately narrowed to a `string` from its initial value:

```
let inventor: number | string = "Hedy Lamarr";

inventor.toUpperCase(); // Ok: string

inventor.toFixed();
//       ~~~~~~~
// Error: Property 'toFixed' does not exist on type 'string'.
```

Conditional Checks

One of the simplest ways to get TypeScript to narrow a variable's value without assigning the value yourself is to write an `if` statement checking the variable for being equal to a known value. TypeScript is smart enough to understand that inside the body of that `if` statement, the variable must be the same type as the known value.

```
let scientist = Math.random() > 0.5
    ? "Rosalind Franklin"
    : 51;
```

```
if (scientist === "Rosalind Franklin") {
    scientist.toUpperCase(); // Ok: string
}

physicist.toUpperCase();
//        ~~~~~~~~~~~
// Error: Property 'toUpperCase' does not exist on type 'string |
number'.
//   Property 'toUpperCase' does not exist on type 'number'.
```

TypeScript also recognizes the `typeof` operator in narrowing down variable types:

```
if (typeof scientist === "string") {
    scientist.toUpperCase(); // Ok: string
}
```

Logical negations from `!` and `else` statements work as well:

```
if (!(typeof scientist === "string")) {
    scientist.toFixed(); // Ok: number
} else {
    scientist.toUpperCase(); // Ok: string
}
```

The above can be rewritten with a ternary statement, which is also supported for type narrowing:

```
typeof scientist === "string"
    ? scientist.toUpperCase() // Ok: string
    : scientist.toFixed(); // Ok: number
```

Narrowing with conditional logic shows TypeScript’s type checking logic mirroring good JavaScript coding patterns. If a variable might be one of several types, you’ll generally want to check its type for being what you need. TypeScript is forcing us to play it safe with our code. Thanks, TypeScript!

TypeScript's type checker recognizes several more forms of narrowing that we'll see in later chapters.

Summary

In this chapter, I went over how TypeScript can deduce more specific ("narrow") type information from how your code is structured:

- How a union types represent values that could be one of two or more types
- Explicitly indicating union types with type annotations
- How type narrowing
- Triggering type narrowing Conditional checks causing TypeScript to narrow specific values

Chapter 4. Literals

TODO: How many of your examples would still work if they were written around built-in JS objects? If you can find common objects that frontend and Node devs would both know (Date?), that's ideal. It allows devs to build understanding from a more concrete place, I think.

Chapter 3 introduced the concept of union types and narrowing, allowing you to work with values that may be two or more potential types. I'll now go the opposite direction by introducing *literal* types: more specific versions of primitive types.

Literal Types

Take this `philosopher` variable:

```
const philosopher = "Hypatia";
```

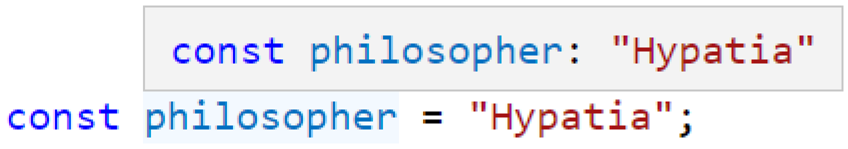
What type is `philosopher`?

At first glance, you might say `string`—and you'd be correct. `philosopher` is indeed a `string`.

But! `philosopher` is not just any old `string`. It's specifically the value `"Hypatia"`. Therefore, the `philosopher` variable's type is technically `"Hypatia"` itself as well.

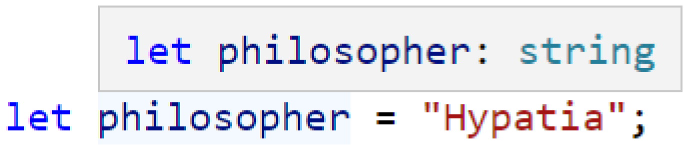
Such is the concept of a *literal*: a value that is known to be a distinct instance of a primitive.

If you declare a variable as `const` and directly give it a literal value, TypeScript will understand the variable to be that literal value as a type. This is why, when you hover a mouse over a `const` variable in an IDE such as VS Code, it will show you the variable's type as that literal (Figure 4-1) instead of the more general primitive (Figure 4-2).



```
const philosopher: "Hypatia"
const philosopher = "Hypatia";
```

Figure 4-1. TypeScript reporting a const variable as being specifically its literal type.



```
let philosopher: string
let philosopher = "Hypatia";
```

Figure 4-2. TypeScript reporting a let variable as being generally its primitive type.

You can think of each *primitive* type as a *union* of every possible matching *literal* value. Most primitive types have a theoretically infinite number of literal types, but not all. Out of some common ones you'll find in typical TypeScript code:

- `boolean`: Just `true | false`
- `null` and `undefined`: both just have one literal value, themselves

- number: `0 | 1 | 2 | ... | 0.1 | 0.2 | ...`
- string: `"" | "a" | "b" | "c" | ... | "aa" | "ab" | "ac" | ...`

Union type annotations can mix and match between literals and primitives. A representation of a lifespan, for example, might be represented by any `number` *or* one of a couple known edge cases:

```
let lifespan: number | "ongoing" | "unknown"; // Type definition

lifespan = 89; // Ok
lifespan = "ongoing"; // Ok

lifespan = true;
// Error: Type 'true' is not assignable to
// type 'number | "ongoing" | "unknown"'
```

Literal Assignability

Literal types are *subsets* of their general primitive types. A value that is known to be a different literal, or only as a general primitive, may not be assigned to a variable that only allows a different literal.

In this example, `specificallyAda` is declared as being of the literal type `"Ada"`, so the types `"Byron"` and `string` are not assignable to it:

```
let specificallyAda: "Ada"; // Type definition

specificallyAda = "Ada"; // Ok

specificallyAda = "Byron";
// Error: Type '"Byron"' is not assignable to type '"Ada"'.

let someString = "";

specificallyAda = someString;
// Error: Type 'string' is not assignable to type '"Ada"'.
```

Literal types are, however, allowed to be assigned to their corresponding primitive types. A specific literal `string` is still generally a `string`, after all.

In this code example, the value `":)"`, which is of type `":)"`, is being assigned to the `someString` variable previously inferred to be of type `string`.

```
someString = ":)";
```

Who would have thought a simple variable assignment would be so theoretically intense?

Strict Null Checking

The power of narrowed unions with literals is particularly visible when working with potentially undefined valuables, an area of type systems known as *strict null checking*. TypeScript is part of a surge of modern programming languages that utilizes strict null checking to fix the dreaded *"billion dollar mistake"*.

The Billion Dollar Mistake

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965… This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*
>
> —Tony Hoare, 2009

The *"billion dollar mistake"* is a catchy industry term for many type systems allowing values such as `null` to be used in places that require a different type. In languages without strict null checking, code like this example that assign `null` to a `string` is allowed:

```
const firstName: string = null;
```

If you’ve previously worked in a typed language such as C++ or Java that suffers from the billion dollar mistake, it may be surprising to you that some languages don’t allow such a thing. If you’re never worked in a language with the strict null checking before, it may be surprising that some languages allow such the billion dollar mistake in the first place!

The TypeScript compiler has an `strictNullChecks` option to toggle whether strict null checking is enabled. With the option set to `false`, the following code is considered totally type safe:

```
let nameMaybe = Math.random()
    ? "Tony Hoare"
    : undefined;

nameMaybe.toLowerCase();
```

Roughly speaking, enabling `strictNullChecks` adds `| null | undefined` to every type in your code, thereby allowing any variable to receive `null` or `undefined`.

TypeScript best practice is generally to enable strict null checking. Doing so helps prevent crashes and eliminates the billion dollar mistake.

With strict null checking enabled, TypeScript sees the potential crash in the code snippet:

```
let nameMaybe = Math.random()
    ? "Tony Hoare"
    : undefined;

nameMaybe.toLowerCase();
// Error: Object is possibly 'undefined'.
```

Truthiness Narrowing

TODO: Maybe define truthiness. Yes, they should know but it won't hurt to redefine it. ‘犊TypeScript can also narrow a variable's type from a truthiness check if only some of its potential values may be truthy. In the following snippet, `geneticist` is of type `string | undefined`, and because `undefined` is always falsy, TypeScript can deduce that it must be of type `string` within the `if` statement's body:

```
let geneticist = Math.random() > 0.5
    ? "Barbara McClintock"
    : undefined;

if (geneticist) {
    geneticist.toUpperCase(); // Ok: string
}

geneticist.toUpperCase();
// Error: Object is possibly 'undefined'.
```

Logical operators that perform truthiness checking work as well, namely `&&` and `?.`:

```
geneticist && geneticist.toUpperCase(); // Ok: string
geneticist?.toUpperCase(); // Ok: string
```

Unfortunately, truthiness checking doesn't go the other way. If all we know about a `string | undefined` value is that it's falsy, that doesn't tell us whether it's an empty string or `undefined`.

Here, `placeholder` is of type `false | string`, and while it can be narrowed down to just `string` in the `if` statement body, the `else` statement body knows it can still be a string if it's `""`:

```
let biologist = Math.random() > 0.5 && "Rachel Carson";

if (!biologist) {
    biologist.toUpperCase(); // Not ok: type `false | string`
           // ~~~~~~~~~~~
           // Error: Property 'toUpperCase' does not
           // exist on type 'string | false'.
           //   Property 'toUpperCase' does not
```

```
                //   exist on type 'false'.
}
```

Implicit Union Type Truthiness

TODO: This paragraph doesn’t read smoothly to me. I would focus on the JavaScript behavior first, what happens when you define but don’t assign? Then, how does TypeScript reconcile that undefined value against a strongly typed declaration? Then, show how the compiler protects with an error message. ‗‘犊If you declare a variable with a type that doesn’t include `undefined` *and no initial value, TypeScript is smart enough to understand that the variable is* `undefined` *until a value is assigned. It will even give you a specialized error message letting you know if you try to access a member before assigning a value:*

```
let mathematician: string;

mathematician.length;
// Error: Variable 'mathematician' is used before being assigned.

mathematician = "Mark Goldberg";
mathematician.length; // Ok
```

Summary

In this chapter, you saw how TypeScript handles specific “literal” values within primitives:

- The difference between `const` literals and `let` primitives
- The “Billion Dollar Mistake” and how TypeScript handles strict null checking
- Using explicit `| undefined` to represent values that might not exist
- Implicit `| undefined` for unassigned variables

Part II. Features

Chapter 5. Functions

Function arguments

In one end, out the other

As a return type

In Chapter 4, you saw how to use type annotations to annotate values of variables without an initial value. Now, you’ll see how to do the same with functions.

Function Parameters

Take the following `sing` function that takes in a `song` parameter and logs it:

```
function sing(song) {
  console.log(`␣椕{song} ␣栩;
}
```

What type is `song`? Is it a `string`? Is it an object with an overridden `toString()` method? *Who knows?!*

Without explicit type information declared, we may never know — TypeScript will consider it to be the `any` type, meaning the parameter's type could be anything.

As with variables, TypeScript allows you to declare the type of function parameters with a type annotation. Now we can use a `: string` to tell TypeScript that the `song` parameter is of type `string`:

```
function sing(song: string) {
  console.log(`␣楸{song} ␣栩;
}
```

Much better!

In theory, you don't *need* to add proper type annotations to function parameters for your code to be valid TypeScript syntax. TypeScript might yell at you with type errors but the emitted JavaScript will still run.

The following snippet will still convert from TypeScript to JavaScript even though the `song` parameter is missing a type:

```
function singAt(song, volume: number) {
//              ~~~~
// Error: Parameter 'song' implicitly has an 'any' type.
  console.log(`${song} at ${volume}`);
}

singAt("Run the World", "three");
//                      ~~~~~~~
// Error: Argument of type 'string' is not
// assignable to parameter of type 'number'.
```

Chapter ?? will cover how to configure TypeScript's complaints about parameters that are implicitly of type `any` the way `song` is.

Required Parameters

TypeScript assumes that all parameters to a function are required. If a function is called without the right number of parameters, TypeScript will protest. TypeScript's parameter counting will come into play both if a function is called with *too few* or *too many* parameters.

This `singTwo` function requires two parameters, so passing one parameter and passing three parameters are both not allowed:

```
function singTwo(first: string, second: string) {
  console.log(`${first} / ${second}`);
}

singTwo("Ball and Chain");
//      ~~~~~~~~~~~~~~~~
// Error: Expected 2 arguments, but got 1.

singTwo("I Will Survive", "Higher Love"); // Ok

singTwo("Go Your Own Way", "The Chain", "Dreams");
//                                      ~~~~~~~~
// Error: Expected 2 arguments, but got 3.
```

Enforcing that required parameters be provided to a function helps enforce type safety by making sure all expected arguments exist inside the function.

Optional Parameters

Recall that in JavaScript, if a function parameter is not provided, its argument value inside the function defaults to `undefined`. Sometimes function parameters are not necessary to provide, and the intended use of the function is for that `undefined` value. TypeScript allows annotating a parameter as optional by adding a `?` before the `:` in its type annotation.

Optional parameters don't need to be provided to function calls. Their types therefore always have `| undefined` added as a union type.

In the following `announceSong` function, the `singer` parameter is marked optional, so while it doesn't need to be provided, its type is `string | undefined`:

```
function announceSong(song: string, singer?: string) {
  console.log(`ㄩ栓ong: ${song}`);

  if (singer) {
    console.log(`F⁖inger: ${singer}`);
  }
}

announceSong("Greensleeves");
announceSong("Chandelier", "Sia");
```

These optional parameters are always implicitly able to be `undefined`. In the above code, `singer` starts off as being of type `string | undefined`, then is narrowed to just `string` by the `if` statement.

Optional parameters are not the same as parameters with union types that happen to include `| undefined`. Parameters that aren't marked as optional with a `?` must always be provided, even if the value is explicitly `undefined`.

```
function announceSongBy(song: string, singer: string | undefined)
{ /* ... */ }

announceSongBy("Greensleeves");
// Error: Expected 2 arguments, but got 1.

announceSongBy("Chandelier", "Sia"); // Ok
```

Default Parameters

Some optional parameters are given a default value with a `=` and value in their declaration. For these optional parameters, because a value is provided by default, the type does not implicitly have the `| undefined` union added on inside the function. However, because they are optional, they are still allowed to be called with a missing or `undefined` argument.

TypeScript's type inference works similarly for default function parameter values as it does for initial variable values: if they exist, you don't need an explicit type annotation. TypeScript can infer what type the parameter is its default value.

In the following `rateSong` function, `rating` is inferred to be of type `number`, but is an optional `number | undefined` when the function is called:

```
function rateSong(song: string, rating = 0) {
  console.log(`${song} gets ${rating}/5 ☆`);
}

rateSong("Photograph"); // Ok
rateSong("Set Fire to the Rain", 5); // Ok

rateSong("At Last!", "100");
//                   ~~~~~
// Error: Argument of type '"100"' is not assignable
// to parameter of type 'number | undefined'.
```

Rest Parameters

Some functions in JavaScript are made to be called with any number of arguments. The `...` spread operator may be placed on the last parameter in a function declaration to indicate any “rest” arguments passed to the function starting at that parameter should all be stored in a single array.

TypeScript allows declaring the types of these rest parameters similarly to regular parameters, except with the `[]` syntax added on to indicate it’s an array of arguments.

Here, `singAllTheSongs` is allowed to take zero or more arguments of type `string` for its `songs` rest parameter:

```
function singAllTheSongs(singer: string, ...songs: string[]) {
  for (const song of songs) {
    console.log(`ս楙{song} Ϝ..{singer}`);
  }
}

singAllTheSongs("Alicia Keys"); // Ok
singAllTheSongs("Lady Gaga", "Bad Romance", "Just Dance", "Poker
Face"); // Ok

singAllTheSongs("Halsey", 100);
//                        ~~~
```

```
// Error: Argument of type 'number' is not
// assignable to parameter of type 'string'.
```

> **NOTE**
>
> Arrays are covered in Chapter 6.

Return Types

TypeScript is smart: if it understands all the possible values returned by a function, it'll know what type the function returns. In this example, `singSongs` is understood by TypeScript to return a `number`:

```
function singSongs(songs: string[]) {
  for (const song of songs) {
    console.log(`␣楙{song} ␣栩;
  }

  return songs.length;
}
```

However, there are a few cases where it's useful to explicitly declare the return type of a function with a type annotation:

- You might choose to add one to remind yourself to return the right value from the function.
- TypeScript will refuse to try to reason through return types of recursive function.
- We'll see in later chapters how explicit type annotations can sometimes explicitly tell TypeScript information it wouldn't have inferred normally.

Function declaration return type annotations are placed after the `)` following the list of parameters.

For a function declaration, that falls just before the `{`.

```
function singSongsRecursive(songs: string[], count = 0): number {
  console.log(`␣䏉{songs[0]} ␣䏊;
  return songs.length ? singSongsRecursive(songs.slice(1), count
+ 1) : count;
}
```

For arrow functions (also known as lambdas), that falls just after the =>:

```
const singSongsRecursive = (songs: string[], count = 0): number
=>
  songs.length ? singSongsRecursive(songs.slice(1), count + 1) :
count;

const singSongsRecursiveWithLog = (songs: string[], count = 0):
number => {
  console.log(`␣䏉{songs[0]} ␣䏊;
  return songs.length ? singSongsRecursive(songs.slice(1), count
+ 1) : count;
};
```

Function Types

JavaScript allows us to pass functions around as values. That means we need a way to declare the type of a parameter or variable meant to hold a function.

Function type syntax looks very similar to an arrow function:

1. (left parenthesis
2. Comma-delineated list of parameters with their types
3.) right parenthesis
4. =>
5. The return type of the function

```
let callback: (songs: string[], count?: number) => number;
```

For example, the following `runOnSongs` snippet declares the type of a `singer` to be a function that takes in a `song: string` and returns nothing:

```
const songs = ["Juice", "Shake It Off", "What's Up"];

function runOnSongs(singer: (song: string) => void) {
  songs.forEach(singer);
}

function sing(song: string) {
  console.log(`␣楙{song} ␣栩;
}

runOnSongs(sing);
```

Function Type Parentheses

Function types may be placed anywhere that another type would be used. That includes array types and union types. Because function types include a space, they may need to be surrounded with parentheses to distinguish what's the individual function type.

In array types, parentheses may be used to indicate which part of an annotation is the function return or the surrounding array type:

```
// Type is a function that returns an array of strings
let createStrings: () => string[];

// Type is an array of functions that each return a string
let stringCreators: (() => string)[];
```

In union types, parentheses may be used to indicate which part of an annotation is the function return or the surrounding union type:

```
// Type is a function that returns a union: string | undefined
let givesStringOrUndefined: () => string | undefined;

// Type is either undefined or a function that gives a string
let maybeGivesString: (() => string) | undefined;
```

Parameter Type Inferences

It would be cumbersome if we had to declare parameter types for every function we write, including inline functions used as parameters. Fortunately, TypeScript can infer function parameter types if it knows the type of the place they're used.

Functions set as values for previously declared variables don't need to have their types declared:

```
let singer: (song: string) => void;

singer = function (song) {
  console.log(`␣楙{song} ␣栩;
};
```

Functions immediately passed to parameters have function parameter types inferred as well:

```
const songs = ["Call Me", "Jolene", "The Chain"];

// song: number
// index: number
songs.forEach((song, index) => {
  console.log(`${song} is at index ${index}`);
});
```

Void Returns

Some functions aren't meant to return any value. They either have no `return` statements or only have `return` statements that don't return a value. TypeScript allows using a `void` keyword to refer to the return type of a function that returns nothing.

This `logSong` function is declared as returning `void`, so it's not allowed to return a value:

```
function logSong(song: string | undefined): void {
  if (!song) {
    return; // Ok
  }

  console.log(`ᴗ楙{song} ᴗ栩;

  return true;
  // Error: Type 'boolean' is not assignable to type 'void'.
}
```

`void` is useful when used as the return type in a function type declaration. For example, this `songLogger` variable represents a function that takes in a `song: string` and doesn't return a value:

```
  let songLogger: (song: string) => void;

  songLogger = (song) => {
    console.log(`ᴗ楙{songs} ᴗ栩;
  };

  songLogger("Heart of Glass"); // Ok
```

Note that although JavaScript functions all return `undefined` by default if no real value is returned, `void` is not the same as `undefined`. `void` means the return type of a function will be ignored, while `undefined` is a literal value to be returned. Functions don't need to actually return `void` in order to be used in locations declared to be a function type with a `void`.

For example, the built-in `forEach` method on Arrays takes in a callback that returns `void`, but the functions provided to `forEach` can return any value they want. `records.push(record)` in the below `saveRecords` function returns the length of the new array, yet is still allowed to be the returned value for the arrow function passed to `records.forEach`:

```
  const records: string[] = [];

  function saveRecords(records: string[]) {
    records.forEach(record => records.push(record));
  }

  saveRecords(['21', 'Come On Over', 'The Bodyguard'])
```

The `void` type is almost exclusively used to declare the return types of functions in TypeScript. Remember, it's an indication that a function's returned value isn't meant to be used, not a value that can itself be returned.

Summary

In this chapter, you saw how a function's parameters and return types can be inferred or explicitly declared in TypeScript:

- Parameters should generally have type annotations that indicate what type they are.
- Optional function parameters without a default value should be marked optional with a `?`.
- `...` rest parameters need to be declared as arrays, as that's what they are
- Function return types can generally be inferred, but may optionally be declared with a type annotation.
- Exception: if the function is recursive, TypeScript will not infer its return type.
- A function type may be declared in the type system like `() =>`, along with its parameters' types and return type > ☝ > TODO: This doesn't help me picture what you're describing.
- Void returns, which are meant for functions that don't return a usable value

Chapter 6. Arrays

Arrays and tuples

One flexible and one fixed

Choose your adventure

JavaScript arrays are wildly flexible and can hold any mixture of values inside:

```
const items = [true, null, undefined, 42];

items.push("even", ["more"]);
```

In practice, though, most JavaScript programs keep the types of data inside any particular array consistent. If an array starts with just one type, it would be confusing to sometimes add another type to it.

TypeScript respects this best practice by remembering what type of data is initially inside an array, and only allowing the array to operate on that kind of data.

In this example, TypeScript knows an array initially contains `string` types, so while adding more `string` typed values is allowed, adding any other type of data is not:

```
const warriors = ["Artemisia", "Boudica"];

// Ok: "Zenobia" is a string
warriors.push("Zenobia");

warriors.push(true);
//            ~~~~
// Argument of type 'boolean' is not assignable to parameter of
type 'string'.
```

You can think of TypeScript's understanding of an array's type from its initial members as a similar area of logic to how it understands variable types from their initial values. TypeScript generally tries to understand the intended types of your code from how values are declared, and arrays are no exception.

Array Types

As with other variable declarations, variables containing arrays don't need to have an initial value. They can start off undefined and receive a value later.

TypeScript will want you to let it know what types of values are meant to go in the array by giving the variable a type annotation. The type annotation for an array requires the type of items in the array followed by a `[]`:

```
let arrayOfNumbers: number[];

arrayOfNumbers = [4, 8, 15, 16, 23, 42];
```

If you don't include that type annotation on a variable initially set an empty array, TypeScript will treat the array as implicitly `any[]`, meaning it can receive any content. As with non-array variables, we don't like implicit `any`s. They partially negate the benefits of TypeScript's type checker.

```
// Type is any[]
let arrayOfNothing = [];

// ...no rules!!
arrayOfNothing.push(123, "something", null);
```

Multi-Dimensional Arrays

A 2D array, or an array of arrays, will have two `[]`s:

```
let arrayOfArraysOfNumbers: number[][];

arrayOfArraysOfNumbers = [
  [1, 2, 3],
  [2, 4, 6],
  [3, 6, 9],
];
```

A 3D array, or an array of arrays of arrays, will have three `[]`s. 4D arrays have four `[]`s. 5D arrays have five `[]`s. You can guess where this is going for 6D arrays and more.

These multi-dimensional array types don't introduce any new concepts to array types. Think of a 2D array as taking in the original type, which just so happens to have `[]` at the end, and adding a `[]` after it.

This `arrayOfArraysOfNumbers` array is of type `number[][]`, which is also representable by `(number[])[]`:

```
// Type is number[][]
let arrayOfArraysOfNumbers: (number[])[];
```

Union Type Arrays

JavaScript arrays can contain all sorts of different types of values, as you saw in the beginning of this chapter. You can use a union type to indicate that each member of an array is either one type or another.

When using array types with unions, parenthesis may be used to indicate which part of an annotation is the contents of the array or the surrounding union type:

```
// Type is either a number or an array of strings
let numberOrStrings: number | string[];

// Type is an array of items that are each either a number or a
string
let stringCreators: (number | string)[];
```

TypeScript will understand from an array's declaration that it is a union type array if it contains more than one type of element. In other words, the type of an array is the list of all possible types that could exist in that array.

Here, `namesMaybe` is `(string | undefined)[] because it has both `string` values and an `undefined` value:

```
// Type is (string | undefined)[]
const namesMaybe = [
  "Aqualtune",
  "Blenda",
  undefined,
];
```

Array Members

TypeScript understands typical index-based access for retrieving members of an array to give back an item of that an array's type.

This `defenders` array is of type `string[]`, so `defender` is a `string`:

```
const defenders = ["Clarenza", "Dina"];

// Type: string
const defender = defenders[0];
```

Members of union typed arrays are themselves that same union type.

Here, `soldiersOrDates` is of type `(Date | string)[]`, so the `soldierOrDate` variable is of type `Date | string`:

```
const soldiersOrDates = ["Deborah Sampson", new Date(1782, 6,
3)];

// Type: Date | string
const soldierOrDate = soldiersOrDates[0];
```

Caveat: Unsound Members

This code gives no complaints with the default TypeScript compiler settings:

```
function withItems(items: string[]) {
  console.log(items[9001].length);
}

withItems([]);
```

We can see that it'll crash at runtime with `` `*Cannot read property length of undefined*` ``, but TypeScript intentionally does not make sure retrieving array members exist. `items[9001]` in the code snippet above is of type `string`, not `undefined`.

Array members are an example of an area where the TypeScript type system is known to be technically *unsound*: it can get types mostly right, but sometimes it messes up.

NOTE

TypeScript does have a `--noUncheckedIndexedAccess` flag that makes array lookups more restricted and type-safe, but it's quite strict and most projects don't use it. I'll discuss it more in *Chapter ??*.

Spreads and Rests

Remember `...` rest parameters for functions from Chapter 5? Rest parameters and array spreading, both with the `...` operator, are key ways to interact with arrays in JavaScript. TypeScript understands both of them.

Spreads

Arrays can be joined together using the `...` spread operator. TypeScript understands the result array will contain values that can be from either of the input array(s).

If the input arrays are the same type, the output array will be that same type:

If two arrays of different types are spread together to create a new array, the new array will be understood to be a union type array of elements that are either of the two original types:

```
const soldiers = ["Harriet Tubman", "Joan of Arc", "Lozen"];
const soldierAges = [90, 19, 49];

const conjoined = [...soldiers, ...soldierAges];
// Type is (number | string)[]
```

Rests

One nice part of the design of JavaScript function rest parameters is that they work seamlessly with array values in code. An array may be passed directly as the rest parameter for a function that accepts one, and TypeScript's type checker will make sure those types match up.

In the following snippet, any number of `string`s may be passed to `announceNames`, but not any other type:

```
function announceNames(...names: string[]) {
  console.log(names.join(","));
}

const warriors = ["Grace O'Malley", "Nakano Takeko"];
```

```
announceNames(...warriors); // Ok

const warriorAges = [72, 21];

announceNames(...warriorAges);
// Error: Argument of type 'number' is not assignable to
parameter of type 'string'.
```

Tuples

Although JavaScript arrays may be any size in theory, it is sometimes useful to use an array of a fixed size -also known as a “tuple"-. Tuple arrays often have a specific known type at each index which may be more specific than a union type of all possible members of the array.

TypeScript allows describing tuple array types with `[`, a comma-separated list of the type of each member in the array, and `]`.

Here, the array `yearAndWarrior` is declared as having a `number` at index 0 and a `string` at index 1:

```
let yearAndWarrior: [number, string];

yearAndWarrior = [530, "Tomyris"]; // Ok

yearAndWarrior = [false, "Tomyris"];
//                ~~~~~
// Error: Type 'boolean' is not assignable to type 'number'.
```

Tuples are often used in JavaScript alongside array destructuring to be able to assign multiple values at once, such as setting two variables to initial values based on a single condition.

For example, TypeScript recognizes here that `year` is always going to be a `number` and `warrior` is always going to be a `string`:

```
// year type: number
// warrior type: string
let [year, warrior] = Math.random() > 0.5
  ? [272, "Archidamia"]
  : [1858, "Rani of Jhansi"];
```

Tuple Assignability

Tuple types are treated by TypeScript as more specific than variable length array types. That means variable length array types aren't assignable to tuple types.

Here, although we as humans may see `temp1` as having `[boolean, number]` inside, TypeScript infers it to be the more general `(boolean | number)[]` type:

```
// Type: (boolean | number)[]
const pairLoose = [false, 123];

const pairTuple: [boolean, number] = temp1;
//    ~~~~~~~~~
// Error: Type '(number | boolean)[]' is not
// assignable to type '[boolean, number]'.
//   Target requires 2 element(s) but source may have fewer.
```

Tuples of different length are also not assignable to each other, as TypeScript includes knowing how many members are in the tuple in tuple types.

Here, `temp2` must have exactly two members, so although `temp1` starts off the same, `temp1`'s third member prevents it from being assignable to `temp2`:

```
const three: [boolean, number, string] = [false, 1663,
"Nzingha"];

const two: [boolean, number] = temp1;
//    ~~~~~
// Error: Type '[boolean, number, string]' is
// not assignable to type '[boolean, number]'.
//   Source has 3 element(s) but target allows only 2.
```

TIP

The `length` member of a flexible size array is always the `number` primitive, whereas a tuple's `length` member is the number literal such as `1` or `2` of how many members exist in the tuple.

Tuples as Rest Parameters

Because tuples have strong typing information about the member at each position of an array, they can be passed as `...` rest parameters to functions This can be useful if you'd like to store arguments for a function in arrays and pass them to the function later.

Here, the `logPair` function's parameters are typed `string` and `number`. Trying to pass in a a value of type `(string | number)[]` as arguments wouldn't be type safe as the order might not match up. However, if TypeScript knows the value to be a `[string, number]` tuple, it understands the values match up:

```
function logPair(name: string, value: number) {
  console.log(`${name} has ${value}`);
}

const pairArray = ["Amage", 1];

logPair(...pairArray);
// Error: A spread argument must either have a tuple type or be
passed to a rest parameter.

const pairTupleIncorrect: [number, string] = [1, "Amage"];

logPair(...pairTupleIncorrect);
// Error: Argument of type 'number' is not assignable to
parameter of type 'string'.

const pairTupleCorrect: [string, number] = ["Amage", 1];

logPair(...pairTupleCorrect); // Ok
```

If you really want to go wild with your rest parameters tuples, you can mix them with arrays to store a list of arguments for multiple function calls. Here, `trios` is an array of tuples, where each tuple also has a tuple for its second member. `trios.forEach(logTrio)` is known to be safe because each `trio` happens to match the parameter types of `logTrio`.

```
function logTrio(name: string, value: [number, boolean]) {
  console.log(`${name} has ${value[0]} (${value[1]}`);
}

const trios: [string, [number, boolean]][] = [
  ["Amanitore", [1, true]],
  ["Æthelflæd", [2, false]],
  ["Ann E. Dunwoody", [3, false]]
];

trios.forEach(logTrio); // Ok
```

Tuple Inferences

TypeScript generally treats created arrays as variable length arrays, not tuples. If it sees an array being used as a variable's initial value or the returned value for a function then it will assume a flexible size array rather than a fixed size tuple.

The following `firstCharAndSize` function is inferred as returning `(number | string)[]`, not `[string, number]`, because that's the type inferred for its returned array literal:

```
// Return type: (number | string)[]
function firstCharAndSize(input: string) {
  return [input[0], input.length];
}

// firstChar type: number | string ☹
// size type: number | string ☹
const [firstChar, size] = firstCharAndSize("Cathay Williams");
```

There are two common ways in TypeScript to indicate that a value should be a more specific tuple type instead of a general array type: explicit tuple types and const assertions.

Explicit Tuple Types

Tuple types may be used in type annotations, such as the return type annotation for a function. If the function is declared as returning a tuple type and returns an array literal, that array literal will be inferred to be a tuple instead a more general variable length array.

This `firstCharAndSizeExplicit` function version explicitly states that it returns a tuple of a `string` and `number`:

```
// Return type: [string, number]
function firstCharAndSizeExplicit(input: string): [string,
number] {
  return [input[0], input.length];
}

// firstChar type: string
// size type: number
const [firstChar, size] = firstCharAndSizeExplicit("Cathay
Williams");
```

Const Asserted Tuples

Typing out tuple types in explicit type annotations can be a pain for the same reasons as typing out any explicit type annotations. It’s extra syntax for you to write and update as code changes.

As an alternative, TypeScript provides an `as const` operator that can be placed after a value to tell TypeScript to go with the most literal, readonly possible form of the value. If placed after an array literal it will indicate that the array should be treated as a tuple.

```
// Type: (number | string)[]
const temp1 = [1157, "Tomoe"];

// Type: readonly [1157, "Tomoe"]
const temp2 = [1157, "Tomoe"] as const;
```

Note that `as const` assertions go beyond switching from flexible sized arrays to fixed size tuples: they also indicate to TypeScript that the tuple is readonly and cannot be used in a place that expects it should be allowed to modify the value.

In this example, `pairMutable` is allowed to be modified because it has a traditional explicit tuple type. However, `` as const`s make the value not assignable to th mutable `pairAlsoMutable ``, and members of the constant `pairConst` not allowed to be modified:

```
const pairMutable: [number, string] = [1157, "Tomoe"];
pairMutable[0] = 1247; // Ok

const pairAlsoMutable: [number, string] = [1157, "Tomoe"] as
const;
//    ~~~~~~~~~~~~~~~
// Error: The type 'readonly [1157, "Tomoe"]' is 'readonly'
// and cannot be assigned to the mutable type '[number, string]'.

const pairConst = [1157, "Tomoe"] as const;
pairConst[0] = 1247;
//        ~
// Error: Cannot assign to '0' because it is a read-only
property.
```

In practice, readonly tuples are convenient for function returns. Returned values from functions that return a tuple are often destructured immediately anyway, so the tuple being readonly does not get in the way of using the function.

This `firstCharAndSizeAsConst` returns a `readonly [string, number]`, but the consuming code only cares about retrieving the values from that tuple:

```
// Return type: readonly [string, number]
function firstCharAndSizeAsConst(input: string) {
  return [input[0], input.length] as const;
}

// firstChar type: string
```

```
// size type: number
const [firstChar, size] = firstCharAndSizeAsConst("Ching Shih");
```

NOTE

`as const` assertions show up later in this book with object types in Chapter 10: Type Modifiers.

Summary

In this chapter, you worked with declaring arrays and retrieving their members:

- Like variables, array types are inferred from their initial values and are not allowed to change
- Array types may be declared by placing a `[]` after the contents of the array
- Arrays of arrays are allowed by placing `[]` after `[]`
- Union type arrays use parenthesis to distinguish what's in the array
- Array members retrieved from indices are known to be the type stored in the array
- These members are technically “unsound”: TypeScript won't know if you're accessing a member at an index that doesn't exist
- Array rests and spreads are understood natively by TypeScript
- Tuples are arrays that have specific types at specific indices
- Tuples often must have types declared: either explicitly or with an `as const` modifier

Chapter 7. Objects and Interfaces

Why only use the

Boring old built-in type shapes…

We can make our own!

So far you've seen how to represent primitive types such as `boolean` and `string` in the type system. Those primitives only scratch at the surface of the complex object shapes JavaScript code commonly uses. TypeScript would be pretty unusable if it weren't able to represent more those objects. This chapter will cover how to describe complex objects shapes and how TypeScript checks their assignability.

Objects

Let’s take a deeper look at how TypeScript reads and understands objects. TypeScript understands that the following `poet` variable’s type is that of an object shape with two member properties: `born`, of type `number`, and `name`, of type `string`:

```
const poet = {
  born: 1830,
  name: "Emily Dickinson"
};

let poetName: string;
poetName = poet.name;
```

The `poetName` variable is allowed to be assigned `poet.name` because TypeScript understands both to be of type `string`.

Reading directly from existing objects is all fine and good, but eventually you’ll want to be able to declare the type of an object separately. You’ll need a way to assign that shape a name and use that name in type annotations.

Interfaces

The most common way to describe an object’s shape in TypeScript is with an *interface*. An interface is a named list of properties that are expected to exist on an object. TypeScript will know anything that’s declared to be of a particular interface’s type will have that interface’s declared members.

The following `Poet` interface describes any object that has those same `born` and `name` member properties:

```
interface Poet {
  born: number;
  name: string;
}
```

Later in code, TypeScript will know anything declared to be of type `Poet` will have those `born` and `name` members:

```
function greetPoet(poet: Poet) {
  // Ok: poet.born is known to be of type `number`
  const born: number = poet.born;

  // Ok: poet.name is known to exist (it's a `string`)
  console.log(`${poet.name}, born in ${born}`);
}

// Ok: the object provided to greetPoet has a born number and a
name string
greetPoet({
  born: 1932,
  name: "Sylvia Plath"
})
```

Interfaces are a powerful concept in TypeScript that allow us to represent the complex custom objects often used in JavaScript projects. They're where we start to truly see TypeScript's power in documenting code and enforcing correct usage.

Interfaces Are Not JavaScript

Interfaces, like type annotations, are not compiled to the output JavaScript. They exist purely in the TypeScript type system.

The above poetry code would compile to roughly this JavaScript:

```
function greetPoet(poet) {
  const born = poet.born;
  console.log(`${poet.name}, born in ${born}`);
}

greetPoet({
  born: 1932,
  name: "Sylvia Plath"
});
```

Because interfaces are purely in the type system, you cannot reference them in runtime code. TypeScript will let you know that you're trying to access something that won't exist at runtime:

```
console.log(Poet);
//          ~~~~
// Error: 'Poet' only refers to a type, but is being used as a
value here.
```

Interfaces exist purely as a development-time construct to help you describe the expected shapes of objects.

Structural Typing

TypeScript's type system is *structurally typed*: meaning any object that happens to satisfy an interface is allowed to be used as an instance of that interface. In other words, when you declare that a parameter or variable is of a particular interface type, you're telling TypeScript that whatever object(s) you use there need to have those properties.

The following `WithFirstName` and `WithLastName` interfaces both only declare a single member of type `string`. The `hasBoth` variable just so happens to have both of theme, so it can be passed in locations that are declared as either of the two interface types:

```
interface WithFirstName {
  firstName: string;
}

interface WithLastName {
  lastName: string;
}

const hasBoth = {
  firstName: "Lucille",
  lastName: "Clifton",
};

function takesWithFirstName(withFirstName: WithFirstName) { /*
... */ }

// Ok: `hasBoth` contains a `firstName` property of type `string`
takesWithFirstName(hasBoth);

function takesWithLastName(withFirstName: WithFirstName) { /* ...
```

```
*/ }

// Ok: `hasBoth` contains a `lastName` property of type `string`
takesWithLastName(hasBoth);
```

TIP

Structural typing is similar to the idea of *duck typing*, which comes from the phrase *"if it looks like a duck and quacks like a duck, it's probably a duck"*.

- Structural typing is when there is a static system checking the type — in TypeScript's case, the type checker.
- Duck typing is when nothing checks object types until they're used at runtime. In other words, *JavaScript* is *duck typed* whereas *TypeScript* is *structurally typed*.

Usage Checking

Passing a previously declared object to a location that requires an instance of an interface requires the object to match the interface's required fields. TypeScript will make sure that each of those required fields exists on the interface.

The following `FirstAndLastNames` interface requires both the `firstName` and `lastName` exist. An object containing both of those is allowed to be used in a variable declared to be of type `FirstAndLastNames`, but an object without them is not:

```
interface FirstAndLastNames {
  first: string;
  last: string;
}

function takesFirstAndLastNames(names: FirstAndLastNames) { }

const hasBoth = {
  first: "Todo First",
  last: "Todo Last",
};
```

```
// Ok
takesFirstAndLastNames(hasBoth);

const hasOnlyOne = {
  name: "Todo Only"
};

takesFirstAndLastNames(hasOnlyOne);
// Error: Argument of type '{ first: string; }' is not
// assignable to parameter of type 'FirstAndLastNames'.
//   Property 'last' is missing in type '{ first: string; }'
//   but required in type 'FirstAndLastNames'.ts
```

Mismatched types between the two are not allowed either. Interfaces specify both the names of required properties and the types those properties are expected to be. If an object's property doesn't match the type the interface declares it to be, TypeScript will report a type checking error.

The following `StartDate` interface expects the `start` member to be of type `Date`. The `hasStartDate` object is causing a type error because its `start` is type `string` instead:

```
interface StartDate {
  start: Date;
}

const hasStartDate: StartDate = {
  start: "0000-00-00",
  // Error: Type 'string' is not assignable to type 'Date'.
}
```

Interface property checking goes by the same type checking rules as variables. As long as a property value is assignable to the interface's declared type, TypeScript will allow it.

Declaration Checking

TypeScript adds in a few more checks when an object is explicitly declared to be of a particular interface type. In addition to checking required field, TypeScript will also make sure no extra fields exist. Therefore, declaring an object to be of an interface’s type is a way of getting the type checker to make sure it has only the expected fields from its interface.

Using the `Poet` interface from earlier in this chapter, this `myPoet` variable is allowed for having all fields match up:

```
// Ok: all fields match what's expected in Poet
const myPoet: Poet = {
  born: 1928,
  name: "Maya Angelou"
};
```

However, this `extraProperty` variable triggers a declaration-specific type error for having an extra property not declared on the `Poet` interface:

```
const extraProperty: Poet = {
    name: "Chell",
    favoriteFood: "potato", /*
    ~~~~~~~~~~~~~~~~~~~~~~
    // Error: Type '{ name: string; favoriteFood: string; }'
    // is not assignable to type 'Poet'.
    //   Object literal may only specify known properties,
    //   and 'favoriteFood' does not exist in type 'Poet'.
*/
};
```

Excess property checks will trigger anywhere a new object is being created in a location that expects it to match an interface — including array members, function parameters, and variables. Banning excess properties is another way TypeScript helps make sure your code is clean and does what you expect. Excess properties not declared in their interface types are often unused code that you the author didn’t mean to include.

Types of Properties

JavaScript objects can be wild and wacky in real world usage, including getters and setters, properties that may not exist, or accepting arbitrary names. TypeScript provide a set of tools for interfaces to help us model that wackiness in the type system.

Optional Properties

Interface properties don't all have to be marked as required. As with function parameters, you can include a `?` before the `:` type in an interface property's type annotation to indicate that it's an optional property.

```
interface Placeholder {
  required: boolean;
  optional?: number;
}
```

Optional properties automatically add the `| undefined` union type added to its type. Keep in mind there is a key addition with marking a property as optional. A property declared as optional with `?` can not exist *and* it can be `undefined`. A property declared as required and `| undefined` must exist, even if the value is `undefined`.

The `optional` property below may be skipped in declaring variables because it has a `?` in its declaration. The `required` property does not have a `?` so it must exist, even if its value is just `undefined`:

```
interface SomeOptionals {
  optional?: string;
  required: string | undefined;
}

// Ok: required is provided as undefined
const hasRequired: SomeOptionals = {
  required: undefined,
};

const missingRequired: SomeOptionals = {};
//    ~~~~~~~~~~~~~~~
// Error: Property 'required' is missing in type
// '{}' but required in type 'SomeOptionals'.
```

Chapter ?? will cover TypeScript's intricate settings around optional properties in more depth.

Readonly Properties

You may sometimes wish to block users of your interface from reassigning members of that interface. TypeScript allows you to add a `readonly` modifier before a member name to indicate that once set, that member should not be set to a different value. These `readonly` members can be read from normally, but not set to anything new.

For example, the `announcement` property in the below `Messenger` interface gives back a `string` when accessed, but causes a type error if assigned a new value:

```
interface Messenger {
    readonly announcement: string;
}

function proclaim(messenger: Messenger) {
    // Ok: reading the message property doesn't attempt to modify
it
    console.log(messenger.announcement);

    messenger.announcement += "?";
    //        ~~~~~~~~~~~~
    // Error: Cannot assign to 'announcement'
    // because it is a read-only property.
}
```

Note that an interface's `readonly` modifier exists only in the type system, and only applies to the usage of that interface. It won't apply to an object unless that object is used in a location that declares it to be of that interface.

In this continuation of the `proclaim` example, the `announcement` property is allowed to be modified outside of the function because its parent object isn't explicitly used as a `Messenger` until inside the function:

```
const messengerIsh = {
  announcement: "Hello, world!",
```

```
}

// Ok: messengerIsh is an inferred object type with announcement,
not a Messenger
messengerIsh.announcement += "!";

// Ok: proclaim takes in Messenger, which happens to
// be a more specific version of messengerIsh's type
proclaim(messengerIsh);
```

Readonly interface members are a good way to make sure areas of code don't unexpectedly modify objects they're not meant to. I'll show you more nifty modifications around readonly properties in *Chapter ??*.

Functions and Methods

It's very common in JavaScript for object members to be functions. TypeScript therefore allows declaring interface members as being the function types previously covered in Chapter 5.

TypeScript provides two ways of declaring interface members as functions:

- **Method** syntax, like `member(): void`: declaring that a member of the interface is a function intended to be called as a member of the object.
- **Property** syntax, like `member: () => void`: declaring that a member of the interface is equal to a standalone function.

The two declaration forms are an analogy for the two ways you can declare a JavaScript object as having a function.

While both `method` and `property` members below are functions that may be called with no parameters and return a `string`:

```
interface HasBothFunctionTypes {
  property: () => string;
  method(): string;
}

const hasBoth: HasBothFunctionTypes = {
  property: () => "",
  method() {
    return "";
  }
};

hasBoth.property(); // Ok
hasBoth.method(); // Ok
```

Both forms can receive the `?` optional modifier to indicate they don't need to be provided:

```
  interface OptionalReadonlyFunctions {
    optionalProperty?: () => string;
    optionalMethod?(): string;
  }
```

As in JavaScript, while the two may seem similar at first, there are a few subtle and occasionally important differences between the two type system declaration forms.

Property Functions

Think of property functions as being interface members that just so happen to be functions rather than some other type. Property functions may be marked as `readonly` just like any other type of property:

```
  interface HasPropertyFunctions {
    required: () => number;
    sometimesOptional?: () => number;
    sometimesRequired: (() => number) | undefined;
    readonly always: () => number;
  }

  const hasPropertyFunctions: HasPropertyFunctions = {
    required: () => 0,
    sometimesRequired: undefined,
    always: () => 0,
  };
```

TODO: Should end with explanation of why to use property over method but I legitimately don't know. †㊊===== Method Functions

Method functions are a special form of interface members that specifically refer to functions whose `this` scope is not inherently bound to the object. Method functions may be marked as optional but may not be marked as `readonly`. Think of method functions as a specific use case for declaring methods on JavaScript objects.

```
interface HasMethodFunction {
  method(): number;
}

const hasMethodFunction: HasMethodFunction = {
  method() {
    return 0;
  },
};
```

Because method functions and property functions can both be called in many of the same ways, TypeScript will often let you use either implementation with either:

```
const actuallyHasPropertyFunction: HasMethodFunction = {
  method: () => 0, // Ok
};
```

I'll show in Chapter 8 how method functions are also used to describe class members for classes that adhere to interfaces. Later on in *Chapter ??* I'll further expand on that with `this` types that make TypeScript care about how the function is being called.

TODO: Should end with explanation of why to use method over property but I legitimately don't know. †㊊[NOTE]

Example 7-1.

Don't sweat it if you mix these two up, or don't understand the difference. It'll rarely impact your code unless you're being intentional about `this` scoping and which form you choose.

Interface Extensions

Sometimes you may end up with multiple interfaces that look similar to each other. One interface may coincidentally contain all the same members of another interface, with a few extras added on. TypeScript provides a syntax called *interface extensions* to declare an interface as copying all the members of another.

An interface may be marked as extending another interface by adding the `extends` keyword after its name (the "derived" interface), followed by the name of the interface to extend (the "base" interface). Doing so indicates to TypeScript that all objects adhering to the derived interface must also have all the members of the base interface.

In the following example, the `Gymnast` interface extends from `Athlete` and thus requires objects to have at least both `Gymnast`'s `moves` and `Athlete`'s `name` members:

```
interface Athlete {
    name: string;
}

interface Gymnast extends Athlete {
    moves: string[];
}

// Ok
let gymnast: Gymnast = {
    moves: ['TODO'],
    name: "Simone Biles",
};

let missingMoves: Gymnast = {
 // ~~~~~~~~~~~~
 // Error: Property 'areas' is missing in type
 // '{ name: string; }' but required in type 'Gymnast'.
    name: "Anonymous",
}

let actuallyTennis: Gymnast = {
    name: "Serena Williams",
```

```
    strategy: "baseline",
 // ~~~~~~~~~~~~~~~~~~~~
 // Error: Type '{ name: string; strategy: string; }'
 // is not assignable to type 'Gymnast'.
 //   Object literal may only specify known properties,
 //   and 'strategy' does not exist in type 'Gymnast'.
}
```

Interface extensions are a nifty way to avoid having to type out the same code repeatedly across multiple interfaces. They all for declaring object shapes that naturally reflect how some objects in JavaScript are supersets of others.

Implementing Multiple Interfaces

Interfaces in TypeScript are allowed to be declared as extending multiple other interfaces. Any number of interfaces separated by commas may be used after the `extends` keyword following the derived interface's name. The derived interface will receive all members from all base interfaces.

Here, the `GetsBoth` has three methods: one on its own, one from `GivesNumber`, and one from `GivesString`:

```
interface GivesNumber {
  giveNumber(): number;
}

interface GivesString {
  giveString(): string;
}

interface GivesBoth extends GivesNumber, GivesString {
  giveEither(): number | string;
}

function useGetsBoth(instance: GetsBoth) {
  instance.giveEither(); // Type: number | string
  instance.giveNumber(); // Type: number
  instance.giveString(); // Type: string
}
```

By marking an interface as extending multiple other interfaces, you can both reduce code duplication and make it easier for object shapes to be mix-and-match reused across different areas of code.

Summary

This chapter introduced how object types may be described by interfaces:

- Declaring interfaces with object type members
- How objects may be declared as being of an interface's type
- Assignability rules for immediately and later-used objects
- Various interface property types, such as optional and readonly
- Reusing interfaces using composition and inheritance

Next up will be a native JavaScript syntax for setting up multiple objects to have the same properties: classes.

Chapter 8. Classes

Some functional devs

Try to never use classes

Too intense for me.

The world of JavaScript during TypeScript’s creation and release in the early 2010s was quite different from today. Features such as arrow functions and `let`/`const` variables that would later be standardized in ES2015 were still distant hopes on the horizon. Babel was a few years away from its first commit; its predecessor tools such as Traceur that converted newer JavaScript syntax to old hadn’t achieved full mainstream adoption.

TypeScript’s early marketing and feature set were tailored to that world. In addition to its type checking, its transpiler was emphasized — with classes as a frequent example. Nowadays TypeScript’s class support is just one feature amongst many to support all JavaScript language features. TypeScript neither encourages nor discourages class use or any other popular JavaScript pattern.

WARNING

Because classes are a rare example of TypeScript supporting a JavaScript feature before its standardization, TypeScript's default settings for compiling to JavaScript create slightly different outputs than you might expect. See *Chapter ??* for class output options and how to configure them.

Class Methods

TypeScript generally understands class methods using the same rules as how it understands standalone functions. Parameters default to `any` unless given a type or default value; calling the method requires an acceptable number of arguments; return types can generally be inferred if the function is not recursive.

This code snippet defines a `TodoName` class with a `logValue` class method that takes in a single required parameter of type `number`:

```
class TodoName {
    logValue(value: number) {
        console.log("Value:", value);
    }
}

new TodoName().logValue(9001); // Ok

new TodoName().logValue();
             // ~~~~~~~~~~
             // Error: Expected 1 arguments, but got 0.
```

TypeScript recognizes the JavaScript syntax of defining a class constructor of a class using the `constructor` keyword for the name of a class method. Class constructors are treated like typical class methods with regards to their parameters.

This `TodoName` constructor also expects its `value: number` parameter to be provided:

```
class TodoName {
    constructor(value: number) {
        console.log("Created with:", value);
    }
}

new TodoName(9001);

new TodoName();
// Error: Expected 1 arguments, but got 0.
```

Class Properties

In order to read from or write to a property on a class in TypeScript, it must be explicitly declared in the class. Class properties are declared using the same syntax as interfaces: the name followed by an optional type annotation.

TypeScript will not attempt to deduce what members may exist on a class from their assignments in a constructor.

In this example, `explicit` is allowed to be accessed on a class because it is explicitly declared as a `number`, while the `this.forgotten` assignment in the constructor is not allowed:

```
class WithProperties {
    explicit: number;

    constructor(value: number) {
        this.explicit = value; // Ok

        this.forgotten = value;
          // ~~~~~~~~~
          // Error: Property 'forgotten' does not
          // exist on type 'WithProperties'.
    }
}
```

Explicitly declaring class properties allows TypeScript to quickly understand what is or is not allowed to exist on instances of classes. Later, when class instances are in use, TypeScript uses that understanding to give a type error if code attempts to access a member of a class instance not known to exist, such as with this continuation’s `forgotten`:

```
const withProperties = new WithProperties(9001);

withProperties.explicit; // Ok

withProperties.forgotten;
               // ~~~~~~~~~~
               // Error: Property 'forgotten' does not
               // exist on type 'WithProperties'.
```

Function Properties

Let’s recap some JavaScript method scoping and syntax fundamentals for a bit, as they can be surprising if you’re not accustomed to them.

JavaScript contains two syntaxes for declaring a member on a class to be a callable function. I’ve already shown the method approach of putting parenthesis after the member name, like `myFunction() {}`. The method approach assigns a function to the class prototype, so all class instances use the same function definition.

```
class WithMethod {
    myMethod() {}
}

new WithMethod().myMethod === new WithMethod().myMethod; // true
```

The other syntax is to declare a property whose value happens to be a function. This creates a new function per instance of the class, which can be useful with `() =>` arrow functions whose `this` scope should always point to the class instance.

This `MessageLogger` class contains a single member of name `log` and type `() => void`:

```
class MessageLogger {
    message: string;
```

```
    constructor(message: string) {
        this.message = message;
    }

    logBound = () => {
        console.log(this.message);
    };

    logUnbound() {
        console.log(this.message);
    }
}

// This will log the `"Oh my!"` message as expected
setInterval(new MessageLogger("Oh my!").logBound);

// This will log whatever `window.message` is (often `undefined`)
setInterval(new MessageLogger("Oh my").logUnbound);
```

TypeScript allows both class function syntaxes and generally treats them roughly the same. The `logUnbound` usage unfortunately will not trigger a type error from the type checker.

> ***TODO(JOSH):*** *I should investigate whether an issue already exists for this. If it exists and is accepting PRs, I should send one. If it was closed, I might want to say why here. If it doesn't exist, I should file one.*

Initialization Checking

With strict compiler settings enabled, TypeScript will check that each non-`undefined` property declared on a class is assigned a value in the constructor.

The following `WithValue` class does not assign a value to its `unused` property, which TypeScript recognizes as a type error:

```
class WithValue {
    immediate = 0; // Ok
    later: number; // Ok (see constructor)
    mayBeUndefined: number | undefined; // Ok

    unused: number;
```

```
  // ~~~~~~
  // Error: Property 'unused' has no initializer
  // and is not definitely assigned in the constructor.

     constructor() {
         this.later = 1;
     }
 }
```

Strict initialization checking is useful because it prevents code from accidentally forgetting to assign a value to a class property. Without it, a class instance could be allowed to access a value that might be `undefined` even though the type system says it can't be.

This example would compile happily if strict initialization checking didn't happen, but the resultant JavaScript would crash at runtime:

```
class MissingInitializer {
    alias: string;
}

new MissingInitializer().alias.length;
// TypeError: Cannot read property 'length' of undefined
```

The billion dollar mistake strikes again!

Optional Properties

Much like interfaces, classes in TypeScript may declare a property as optional by adding a `?` after its declaration name. Optional properties behave roughly the same as properties whose types happen to be a union that includes `| undefined`. Strict initialization checking won't mind if they're not explicitly set in their constructor.

> ***NOTE:*** *This section's content is likely to change based on discussion around TypeScript's new* `--strictOptionalProperties` *flag. We should revisit at the end of 2021 when the team hopefully has more info and formed opinions. See* *https://github.com/microsoft/TypeScript/issues/44421*.

Readonly Properties

Again much like interfaces, classes in TypeScript may declare a property as readonly by adding the `readonly` keyword before its declaration name. The `readonly` keyword exists purely within the type system and is removed when compiling to JavaScript

Properties declared as `readonly` may only be assigned initial values in-place or in a constructor. Any other location -including methods on the class itself- may only read from the properties, not write to them.

In this example, the `value` property on the `WithReadonly` class is given an a value in the constructor, but the other uses cause type errors:

```
class WithReadonly {
    readonly value: string;

    constructor() {
        this.value = "I did it";
    }

    oops() {
        this.value = "again";
          // ~~~~~
          // Error: Cannot assign to 'value' because it is a
read-only property.
    }
}

const withReadonly = new WithReadonly();

withReadonly.value = "!";
          // ~~~~~
          // Error: Cannot assign to 'value' because it is a
read-only property.
```

Properties declared as `readonly` with an initial value of a primitive have a slight quirk compared to other properties: they are inferred to be their value's narrowed *literal* type if possible, rather than the wider *primitive*. TypeScript feels comfortable with a more aggressive initial type narrowing because it knows the value won’t be changed later: similar to const variables taking on narrower types than let variables.

In this example, the class properties are both initially declared as 0, so in order to widen one to number, a type annotation is needed:

```
class WithReadonlies {
    readonly explicit: number = 0;
    readonly implicit = 0;

    constructor() {
        this.explicit = 1;

        this.implicit = 1;
     // ~~~~~~~~~~~~~
     // Error: Type '1' is not assignable to type '0'.
    }
}

const withReadonlies = new WithReadonlies();

withReadonlies.explicit; // Type: number
withReadonlies.implicit; // Type: 0
```

Classes and Interfaces

Back in Chapter 7, I showed you how interfaces allow TypeScript developers to set up expectations for object shapes in code. TypeScript allows a class to declare its instances as adhering to an interface by adding the implements keyword after the class name, followed by the name of an interface. Doing so indicates to TypeScript that instances of the class should be assignable to each of those interfaces. Any mismatches would be called out as type errors by the type checker.

In this example, the Zebra class correctly implements the Animal interface by including its property name and method move, but PetRock is missing a move and thus results in a type error:

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

```
    move(distance: number): void;
}

class Zebra implements Animal {
    name: string;

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

    move(distance: number) {
        console.log(`Moving: ${distance}`);
    }
}

class PetRock implements Animal {
   // ~~~~~~~
   // Error: Class 'PetRock' incorrectly implements interface
'Animal'.
   //  Property 'move' is missing in type 'PetRock' but required
in type 'Animal'.
    name = "Rocky";
}
```

Marking a class as implementing an interface doesn't change anything about how the class is used. If the class already happened to match up to the interface, TypeScript's type checker would have allowed its instances to be used in places where an instance of the interface is required anyway. TypeScript won't even infer the types of methods or properties on the class from the interface: if we had added a `move(distance) {}` method to the `PetRock` example above, TypeScript would think the `distance` parameter an implicit `any` unless we gave the `distance` parameter a type annotation.

Implementing an interface is purely a safety check. It does not copy any interface members onto the class definition for you. Rather, implementing an interface allows you to use TypeScript to let you know exactly how a class doesn't match up to an interface where the class is declared, rather than later on with error messages on the class instances. It's similar in purpose to adding a type annotation a variable even though it has an initial value.

Implementing Multiple Interfaces

Classes in TypeScript are allowed to be declared as implementing multiple interfaces. The syntax and rules for doing so are similar to interfaces declaring themselves as extending multiple interfaces. The list of implemented interfaces for a class may be number of interfaces names with commas in-between.

In this example, the both class are required to have at least two members - `numbers`, and `getString`- and the `Empty` class has two type errors for failing to implement each of the interfaces properly:

```
interface HasArray {
    numbers: number[];
}

interface HasFunction {
    getString: () => string;
}

class Placeholder implements HasArray, HasFunction {
    numbers: number[];

    constructor(numbers: number[]) {
        this.numbers = numbers;
    }

    getString() {
        return this.numbers.join(", ");
    }
}

class Empty implements HasArray, HasFunction { }
   // ~~~~~
   // Error: Class 'Empty' incorrectly implements interface
'HasArray'.
   //   Property 'numbers' is missing in type 'Empty' but
required in type 'HasArray'.
   // ~~~~~
   // Error: Class 'Empty' incorrectly implements interface
'HasFunction'.
   //   Property 'getString' is missing in type 'Empty' but
required in type 'HasFunction'.
```

In practice, there may be some interfaces whose definitions make it difficult or impossible to have a class realistically implement both.

```
interface AgeIsANumber {
    age: number;
}

interface AgeIsNotANumber {
    age: () => string;
}

class AsNumber implements AgeIsANumber, AgeIsNotANumber {
    age = 0;
 // ~~~
 // Error: Property 'age' in type 'AsNumber' is not assignable
 // to the same property in base type 'AgeIsNotANumber'.
 //   Type 'number' is not assignable to type '() => string'.
}

class NotAsNumber implements AgeIsANumber, AgeIsNotANumber {
    age() { return ""; }
 // ~~~
 // Error: Property 'age' in type 'NotAsNumber' is not assignable
 // to the same property in base type 'AgeIsANumber'.
 //   Type '() => string' is not assignable to type 'number'.
}
```

Class Extensions

TypeScript adds type checking onto the JavaScript concept of classes extending other classes. To start, any method or property declared on a base class will be available on the derived class.

In this example, `Base` declares an `onBase` method that `Sub` instances may use:

```
class Base {
    onBase() {
        console.log("Hey!");
    }
}

class Sub extends Base {
```

```
    onSub() {
        console.log("You!");
    }
}

const sub = new Sub();
sub.onBase(); // Ok (defined on base)
sub.onSub(); // Ok (defined on derived)

sub.other();
 // ~~~~~
 // Error: Property 'other' does not exist on type 'Sub'.
```

Extension Assignability

TypeScript's structural typing mandates that object satisfying a shape may be used where a value must be that shape. Derived classes *extend* their base class much like how derived interfaces extend base interfaces. Instances of derived classes have all the members of their base class and thus may be used wherever an instance of the base is required. If a base classes doesn't have all the members the derived class does then it can't be used when the more specific derived class is required.

For instance *(ha!)*, instances of the following `BaseWithNumber` class may not be used where instances of its derived `DeriveWithString` are required, but derived instances may be used to satisfy either the base or derived class:

```
class BaseWithNumber {
    myNumber = 0;
}

class DerivedWithString extends BaseWithNumber {
    myString = '';
}

let base: BaseWithNumber;
base = new BaseWithNumber(); // Ok
base = new DerivedWithString(); // Ok

let derived: DerivedWithString;
derived = new DerivedWithString(); // Ok
```

```
derived = new BaseWithNumber();
// Error: Property 'myString' is missing in type
// 'BaseWithNumber' but required in type 'DerivedWithString'.
```

Interestingly, if all the members on a derived class already exist on its base class with the same type, then instances of the base class are still allowed to be used in place of the derived class. TypeScript doesn't mind that the derived and base classes technically have different constructors. As long as the available fields match up, they're considered assignable to each other.

In this example, `DerivedOptional` only adds an optional property to `BaseWithBoolean`, so instances of the base class may be used in place of the derived classes:

```
class BaseWithBoolean {
    age = 0;
    happy = true;
}

class DerivedStructure {
    happy = false;
}

let derived: DerivedStructure;

derived = new DerivedStructure(); // Ok
derived = new BaseWithBoolean(); // Ok
```

> **TIP**
>
> In real world code, derived classes rarely add no new required type information on top of their base class. This structural checking behavior may seem unexpected but doesn't come up very often.

Overridden Constructors

As with vanilla JavaScript, derived classes are not required by TypeScript to define their own constructor. Derived classes without their own constructor implicitly use the constructor from their base class.

If a derived class does declare its own constructor, it may declare any parameters regardless of what its base class requires. As long as the constructor eventually calls `super()` with the correct parameters from the base class —as required in JavaScript already— then TypeScript's type checker will be happy.

In this example, `DerivedCorrect`'s constructor correctly calls the base constructor with a `string` argument, while `DerivedIncorrect` gets a type error for forgetting to make that call:

```
class NeedsString {
    constructor(input: string) {
        console.log(`Received: ${input}`);
    }
}

class DerivedCorrect extends NeedsString {
    constructor() {
        super("Correct!")
    }
}

class DerivedIncorrect extends NeedsString {
    constructor() { }
  // ~~~~~~~~~~~~~~~~~
  // Error: Constructors for derived classes must contain a
'super' call.
}
```

As per JavaScript rules, the constructor of a derived class must call the base constructor before accessing `this` or members on `super`. TypeScript is generally able to detect most cases of a derived class constructor violating that rule, and emit a type error.

The following `HeadStart` class erroneously refers to `this.count` in its constructor before calling to `super()`:

```
class BaseWithMembers {
    count: number;

    constructor() {
```

```
        this.count = 0;
    }

    increment() {
        this.count += 1;
        return this.count;
    }
}

class HeadStart extends BaseWithMembers {
    constructor() {
        this.count = 10;
     // ~~~~
     // Error: 'super' must be called before accessing
     // 'this' in the constructor of a derived class.

        super();

        this.count = 10; // Ok
    }
}
```

WARNING

On a related note, Daniel and Ryan, if you're reading this, ping for PR review please? ᗷ 羸ttps://github.com/microsoft/TypeScript/pull/29374

Overridden Methods

Derived classes may redeclare new methods with the same names as the base class, as long as the method on the derived class method is assignable to the method on the base class. This is because instances of derived classes may be provided in locations typed as the base class; calling methods on those instances must work well regardless of whether it is an instance of the derived class or base class.

TODO: This explanation would be very well-served by a code example

In this example, `GetDoubleStringLength`'s `getLength` is permitted because it has the same parameters and return type as the base

`GetStringLength`'s, while `GetNumbersLength`'s `getLength` causes a type error for having the wrong parameters type.

```
class GetStringLength {
    getLength(input: string) {
        return input.length;
    }
}

class GetDoubleStringLength extends GetStringLength {
    getLength(input: string) {
        return super.getLength(input) * 2;
    }
}

class GetNumbersLength extends GetStringLength {
    getLength(numbers: number[]) {
 // ~~~~~~~~~
 // Error: Property 'getLength' in type 'GetNumbersLength' is not
 // assignable to the same property in base type
'GetStringLength'.
 //   Type '(numbers: number[]) => number' is not
 //   assignable to type '(input: string) => number'.
 //     Types of parameters 'numbers' and 'input' are
incompatible.
 //       Type 'string' is not assignable to type 'number[]'
    }
}
```

Overridden Properties

Derived classes may also explicitly redeclare properties of their base class with the same name, as long as the new type is assignable to the type on the base class. As with overridden methods, derived classes must structurally match up with base classes.

Most derived classes that redeclare properties do so either to make those properties a more specific subset of a type union or to make the properties a type that extends from the base class property's type.

In this example, the base class `NumberMaybe` declares its `value` to be `number | undefined`, while the derived class `NumberDefinitely` declares it as a `number` that must always exist.

```
class NumberMaybe {
    value?: number ;

    constructor(value?: number) {
        this.value = value;
    }
}

class NumberDefinitely extends NumberMaybe {
    value: number;

    constructor(value: number) {
        super(value);

        this.value = value;
    }
}
```

Note that TypeScript applies strict initialization checking to properties regardless of whether they happen to match the name of a property from a base class.

> *TODO: Revisit once https://github.com/microsoft/TypeScript/issues/45268 gets a resolution.*

Adding new type information to the property's union type is not allowed, as doing so would make the derived class property no longer assignable to base class property's type.

In this example, `NumberOrString`'s `value` tries to add `| string` on top of the base class `number` value, causing a type error:

```
class JustNumber {
    value = 0;
}

class NumberOrString extends JustNumber {
    value = Math.random() > 0.5 ? 1 : "...";
   // ~~~~~
   // Error: Property 'value' in type 'NumberOrString' is not
   // assignable to the same property in base type 'JustNumber'.
```

```
 //   Type 'string | number' is not assignable to type 'number'.
 //     Type 'string' is not assignable to type 'number'.
}

const instance: JustNumber = new NumberOrString();

instance.value; // Type should be 'number', not 'string'!
```

Abstract Classes

It can sometimes be useful to create a base class that doesn't itself declare the implementation of some methods, but instead expect a derived class to provide them. Marking a class as abstract is done by adding the `abstract` keyword in front of the class name and in front of any method intended to be abstract. Those abstract method declarations skip providing a body in the abstract base class; they instead are declared the same way an interface would.

In this example, the `Actor` class and its `act` method are marked as `abstract`; its derived classes -`StageActor` and `Wrong`- are expected to implement them:

```
abstract class Actor {
    readonly name: string;

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

    abstract act(): void;
}

class StageActor extends Actor {
    act() {
        console.log("F?
    }
}

class Wrong extends Actor { }
   // ~~~~~
   // Error: Non-abstract class 'Wrong' does not implement
   // inherited abstract member 'act' from class 'Actor'.
```

An abstract class cannot be instantiated directly, as it doesn't have definitions for some methods that its implementation may assume do exist. Only non-abstract ("concrete") classes can be instantiated.

Continuing the `Actor` example:

```
new StageActor(); // Ok

new Actor();
// Error: Cannot create an instance of an abstract class.
```

Extended Abstract Classes

Derived classes may be marked as `abstract` as well. If they are, they won't need to implement all abstract methods from their base class. Abstract derived classes may add their own abstract methods and/or implement as many of their abstract base class methods as they wish.

Here, both `First` and its derived `Second` are abstract, and because `Second` implements `First`'s `apple`, the concrete `Third` classes need to implement `banana` from `First` and `cherry` from `Second`:

```
abstract class First {
    abstract apple(): string;
    abstract banana(): string;
}

abstract class Second extends First {
    apple() {
        return "apple";
    }

    abstract cherry(): string;
}

class ThirdCorrect extends Second {
    banana() { return "banana"; }
    cherry() { return "cherry"; }
}

class ThirdIncorrect extends Second { }
   // ~~~~~~~~~~~~~~
```

```
    // Error: Non-abstract class 'ThirdIncorrect' does not
implement
    // inherited abstract member 'banana' from class 'Second'.
    // ~~~~~~~~~~~~~~
    // Error: Non-abstract class 'ThirdIncorrect' does not
implement
    // inherited abstract member 'cherry' from class 'Second'.
```

Member Visibility

Private class members may only be accessed by instances of that class. JavaScript runtimes enforce that privacy by throwing an error if an area of code not within the class itself tries to access the private method or property.

However, TypeScript's class support predates JavaScript's true # privacy, and while TypeScript supports private class members, it also allows a slightly more nuanced set of privacy definitions on class methods and properties that exist solely in the type system. TypeScript's member visibilities are achieved by adding one of the following keywords before the declaration name of a class member:

- `public` *(default)*: allowed to be accessed by anybody, anywhere
- `protected`: allowed to be accessed only by the class itself and its subclasses
- `private`: allowed to be access only by the class itself

These keywords exist purely within the type system and, unlike `#`, do not change the name of the method or property.

Here, `Base` declares two `public` members, one `protected`, and one `private`. `Derived` is allow to access the `public` and `protected` members but not the `private`:

```
class Base {
    isPublicImplicit = 0;
    public isPublicExplicit = 1;
```

```
    protected isProtected = 2;
    private isPrivate = 3;
}

class Derived extends Base {
    examples() {
        this.isPublicImplicit; // Ok
        this.isPublicExplicit; // Ok
        this.isProtected; // Ok

        this.isPrivate;
     // ~~~~~~~~~~~~~~
     // Error: Property 'isPrivate' is private
     // and only accessible within class 'Base'.
    }
}

new Derived().isPublicImplicit; // Ok
new Derived().isPublicExplicit; // Ok

new Derived().isProtected;
         // ~~~~~~~~~~~
         // Error: Property 'isProtected' is protected
         // and only accessible within class 'Base' and its
subclasses.

new Derived().isPrivate;
         // ~~~~~~~~~~~
         // Error: Property 'isPrivate' is private
         // and only accessible within class 'Base'.
```

The key difference between TypeScript's member visibilities and JavaScript's true private declarations is that TypeScript's exist only in the type system, while JavaScript's exists at runtime. A TypeScript class member declared as `protected` or `public` will compile to the same JavaScript code as if they were declared `public` explicitly or implicitly. As with interfaces and type annotations, visibility keywords are erased when outputting JavaScript.

To declare a member both as `readonly` and with an explicit visibility, the visibility comes first:

```
class TwoKeywords {
    private readonly name: string;
```

```
    constructor() {
        this.name = "two"; // Ok
    }

    log() {
        console.log(this.name); // Ok
    }
}

const two = new TwoKeywords();

two.name = "three";
 //  ~~~~
 // Error: Property 'name' is private and
 // only accessible within class 'TwoKeywords'.
 //  ~~~~
 // Error: Cannot assign to 'name'
 // because it is a read-only property.
```

Parameter Properties

Declaring member properties with their types and then again declaring constructor parameters with the same names and types is a common practice in TypeScript that can feel tedious. TypeScript provides a syntax shortcut called "parameter properties" to declare both declarations at the same time with the same name and value by adding a `readonly` and/or visibility keywords before a constructor parameter. Doing so adds a class member of the same name and type, and assigns the value to the provided argument at the beginning of the constructor.

Here, the `separated` and `together` properties are virtually identical; they're both `public readonly` `string`s assigned to a constructor argument:

```
class PropertyTypes {
    public readonly separated: string;

    constructor(
        separated: string,
        public readonly together: string,
    ) {
```

```
        this.separated = separated;
    }
}

const types = new PropertyTypes("first", "second");

types.separated; // "first"
types.together; // "second"
```

Parameter properties are another rare example of TypeScript adding a non-standard JavaScript feature. Some TypeScript developers prefer using them for the declaration conciseness they bring. Others avoid them for introducing a non-standard JavaScript syntax reliant on TypeScript concepts. It's up to you to decide whether you prefer using them in any particular project.

Static Visibility

JavaScript allows declaring members on a class itself -not its instances- using the `static` keyword. TypeScript allows using the `static` keyword on its own and/or with `readonly` and/or with one of the visibility keywords. The visibility keyword comes first, then `static`, then `readonly`.

Putting them all together:

```
class HasStatic {
    protected static readonly value = "static";

    log() {
        console.log(HasStatic.value); // Ok
    }
}

HasStatic.value;
         // ~~~~~
         // Error: Property 'value' is protected and only
         // accessible within class 'HasStatic' and its
subclasses.
```

Summary

This chapter introduced a plethora of type system features and a rare new syntax around classes:

TODO: The list items should be consistent in form with each other. Here, most of them are just topic names except for the third one, which leads with a verb. I like leading with a verb, as it tells the reader how they can use things. Either form is fine, but please be consistent throughout the list and throughout the book. ꞌ犊- Type rules for declaring and using methods and properties - `readonly` *and optional properties - Implementing interfaces to enforce class instance shapes - Assignability rules for class extensions - Abstract classes and methods - Type system member visibility - Parameter properties*

Chapter 9. Type Modifiers

Types of types from types.

"It's turtles all the way down",

Anders likes to say.

Chapter 9: Generics introduced the first bits of logic in TypeScript's type system with types that can change, be restricted, and default to other types depending on their usage. This chapter will cover several other useful methods of introducing small bits of logic in the type system.

Top Types

A "top" type, or universal type, is a type that can represent any possible type in a system. I've already shown you the most common top type: `any`. All other types can be provided to a location whose type is the top type `any`.

`any`, Again

I’ve previously discussed the `any` type. `any` is generally used when a location is allowed to accept data of an unknown type, such as the parameters to `console.log`:

```
let anyValue: any;
anyValue = 'abc'; // Ok
anyValue = 123; // Ok

console.log(anyValue); // Ok
```

The problem with `any` is that it explicitly tells TypeScript not to perform typechecking on that object’s assignability or members. That lack of safety is useful if you’d like to quickly bypass TypeScript’s type checker, but the disabling of type checking reduces TypeScript’s usefulness for that object.

For example, the nonsense `value.push()` call below definitely will crash, but because `value` is declared as `any`, TypeScript does not emit a type complaint:

```
function logLengthAny(value: any) {
    // No type error...
    console.log(value.push(''));
}

logLengthAny({});
```

If you want to indicate that an object can be anything, TypeScript has a much safer top type you can use: `unknown`.

`unknown`

The `unknown` type in TypeScript is similar to `any` in that it is a top type and thus objects of any type may be passed to locations of type `any`. The key difference with `unknown` is that TypeScript does not allow directly accessing members of it.

Attempting to access a member of an `unknown` value as in the following will cause TypeScript to emit a type error:

```
function logLengthUnknown(value: unknown) {
    console.log(value.length);
    //          ~~~~~~
    // Error: Object is of type 'unknown'.
}

logLengthUnknown({});
```

The only way TypeScript will allow code to access members on an object of type `unknown` is if the object's type is narrowed, such as using `instanceof` or `typeof`.

```
function logLengthUnknownSafely(value: unknown) {
    if (typeof value === "string") {
        console.log(value.length); // Ok
    }
}

logLengthUnknownSafely({});
```

`unknown` is almost always safer to use than `any` because it doesn't allow member accesses until narrowed. You should generally prefer using `unknown` instead of `any` when possible.

Type Guards

The TypeScript type checker's type narrowing understands built-in JavaScript constructs such as `instanceof` and `typeof` checks. That's all fine and good for that limited set of checks, but it gets lost if you wrap the logic with a function.

For example, this `isLongTextBasic` function takes in a string and returns a boolean indicating whether the `text` exists and has a `.length` greater than 7. We as humans can infer that the `text` inside the `if` statement must therefore exist since `isLongTextBasic(text)` returned `true`, but TypeScript does not:

```
function isLongTextBasic(text: string | undefined) {
    return (text?.length ?? 0) > 7;
}

function processTextBasic(text: string | undefined) {
    if (isLongTextBasic(text)) {
        // Text type: string | undefined
        text.toUpperCase();
        // Error: Object is possibly undefined.
    }
}
```

TypeScript allows functions returning a `boolean` to be declared as having a return type that indicates whether a parameter is a particular type. This is referred to as a "user-defined type guard": you the developer are creating your own type guard akin to `instanceof` or `typeof`. User-defined type guards are commonly used to indicate whether an argument passed in as a parameter is a more specific type than the parameter's.

We can change the above example's helper function to have an explicit return type that explicitly states `text is string`. TypeScript will then be able to infer that blocks of code only reachable if `text is string` is `true` must have a `text` of type `string`:

```
function isLongTextGuard(text: string | undefined): text is
string {
    return (text?.length ?? 0) > 7;
}

function processTextGuard(text: string | undefined) {
    if (isLongTextGuard(text)) {
        // Text type: string
        text.toUpperCase(); // Ok
    }
}
```

You can think of a user-defined type guard as returning not just a boolean, but also an indication that the argument was that more specific type.

Note that user-defined type guards do not work in the opposite, falsy direction. If a user-defined function returns `false` then TypeScript will not narrow the type at all. The above `isLongTextGuard` function above, for example, could return `false` even if `text is string` in the case of `text` being a small string such as `""`.

Type Operators

Not all types can be represented using only a keyword or a name of an existing shape. It can sometimes be necessary to create a new type that combines both: performing some transformation on the properties of an existing shape.

`keyof`

JavaScript objects can have members retrieved up using dynamic values, which are commonly (but not necessarily) `` string`s. Representing these keys in the type system can be tricky. Using a catch-all primitive such as `string `` would allow invalid keys for the container object:

```
interface Counts {
    temp1: number;
    temp2: number;
}

function getCountString(counts: Counts, key: string): number {
    return counts[key];
    //     ~~~~~~~~~~~
    // Error: Element implicitly has an 'any' type because
expression
    // of type 'string' can't be used to index type 'Counts'.
    //   No index signature with a parameter of
    //   type 'string' was found on type 'Counts'.
}

getCountString({ temp1: 1, temp2: 2 }, 'temp1'); // Ok
```

```
getCountString({ temp1: 1, temp2: 2 }, 'not valid'); // Ok, but
shouldn't be
```

Another option would be to use a type union of literals for the allowed keys. That would be more accurate in properly restricting to only the keys that exist on the container object:

```
function getCountLiteral(counts: Counts, key: 'temp1' | 'temp2'):
number {
    return counts[key]; // Ok
}

getCountLiteral({ temp1: 1, temp2: 2 }, 'temp1'); // Ok

getCountLiteral({ temp1: 1, temp2: 2 }, 'not valid');
//                                      ~~~~~~~~~~~
// Error: Argument of type '"not valid"' is not
// assignable to parameter of type '"temp1" | "temp2"'.
```

However, what if the interface has dozens or more members? You would have to type out each of those members' keys and keep them up-to-date. What a pain.

TypeScript instead provides a `keyof` operator that takes in an existing type and gives back a union of all the keys allowed on that type. Place it in front of the name of a type wherever you might use a type, such as a type annotation.

Here, `keyof Counts` is equivalent to `'temp1' | 'temp2'` but is much quicker to write out:

```
function getCountKeyof(counts: Counts, key: keyof Counts): number
{
    return counts[key]; // Ok
}

getCountKeyof({ temp1: 1, temp2: 2 }, 'temp1'); // Ok

getCountKeyof({ temp1: 1, temp2: 2 }, 'not valid');
//                                    ~~~~~~~~~~~
// Error: Argument of type '"not valid"' is not
// assignable to parameter of type 'keyof Counts'.
```

`keyof` is also the only way to specify the key of a generic type. Since generic types may change with each usage of their generic construct, asking for the key of one must be done dynamically with `keyof`.

Take this simplified version of the `get` method from the popular library Lodash. It takes in some container object and a `key` name of one of the keys of `Container` to retrieve from `container`. Because the `Key` generic is constrained to be a `keyof Container`, TypeScript knows this function is allowed to return `Container[Key]`:

```
function get<Container, Key extends keyof Container>(container:
Container, key: Key) {
    return container[key];
}

const found = get("placeholder", "length"); // Type: number

const missing = get("placeholder", "not valid");
//                                 ~~~~~~~~~~~
// Error: Argument of type '"not valid"' is not assignable to
// parameter of type 'number | "length" | ... | "matchAll"'.
```

Without `keyof`, there would have been no way to type the generic `key` parameter.

typeof

Another type operator provided by TypeScript is `typeof`. It gives back the shape of the object whose name is provided to it. This can be useful if the type of an object would be annoyingly complex to write out manually.

Here, the `clone` variable is declared as being the same type as `original`.

```
const original = {
    count: 0,
    value: 'temp',
};

let clone: typeof original;
```

```
if (Math.random() > 0.5) {
    clone = { ...original, count: original.count + 1 }; // Ok
} else {
    clone = { ...original, count: 'two' };
    //                       ~~~~~
    // Error: Type 'string' is not assignable to type 'number'.
}
```

TIP

Although the `typeof` *type* operator visually looks like the *runtime* `typeof` operator used to return a string description of an object's type, the two are different.

`keyof typeof`

`typeof` retrieves the type of an object and `keyof` retrieves the allowed keys on a type. TypeScript allows the two keywords to be chained together to succinctly retrieve the allowed keys on an object's type. Putting them together, the `typeof` type operator becomes wonderfully useful for working with `keyof` type operations.

In this example, the `logCount` function is meant to take in one of the keys of the `counts` object. Instead of creating an interface the code uses `keyof typeof` to indicate `key` must be one of the keys on the type of the `counts` object:

```
const counts = {
    temp1: 1,
    temp2: 2,
    temp3: 3,
};

function logCount(key: keyof typeof counts) {
    console.log(counts[key]);
}

logCount('temp1'); // Ok

logCount('unknown');
//       ~~~~~~~~~
```

```
// Error: Argument of type '"unknown"' is not assignable
// to parameter of type '"temp1" | "temp2" | "temp3"'.
```

Type Assertions

TypeScript works best when your code is "strongly typed": all the objects in your code have well-known shapes. Features such as top types and type guards provide ways to wrangle complex code into being understood by TypeScript's type checker. However, sometimes it's not reasonably possible to be 100% accurate in telling the type system how your code is meant to work.

For example, `JSON.parse` intentionally returns the top type `any`. There's no way to inform the type system that a particular string value given to `JSON.parse` should return any particular object type. (As we saw in *Chapter 9: Generics*, adding a generic type to `parse` that is only used once for a return type would violate The Golden Rule of Generics.)

TypeScript provides a syntax for overriding the type system's understanding of an object's type: a "type assertion", also known as a "type cast". On an object that is meant to be a different type, you can place the `as` keyword followed by a type. TypeScript will defer to your assertion and treat the object as that type.

In this snippet, it is visibly certain to a reader that the returned result from `JSON.parse` should be either `[string, string]` or `string[]` because we can see the input value is an array of three strings. The snippet uses `as` casts for two of the lines of code to switch the type from `any` to one of those:

```
// Type: any
JSON.parse(`["a", "b"]`);

// Type: string[]
JSON.parse(`["a", "b"]`) as string[];

// Type: string[]
JSON.parse(`["a", "b"]`) as [string, string];
```

NOTE

TypeScript also allows casting in `.ts` files with a syntax that looks like `<type>item` instead of `item as type`. This predated the popularity of JSX syntax and does not work in `.tsx` files, so it is discouraged.

Error handling is another place where type assertions may come in handy. It is impossible to know what type an Still, if you are absolutely confident that nothing in your code will throw an error other than what you specifically want it to, you can cast the error variable in a `catch` statement to that type.

This snippet assumes that nothing inside `new ValueError(9001)` will throw an error, so the `error` in the `catch` statement is casted to a `ValueError` when its `.value` is needed:

```
class ValueError extends Error {
    readonly name = "ValueError";
    readonly value: number;

    constructor(message: string, value: number) {
        super(message);
        this.value = value;
    }
}

try {
    throw new ValueError("Oh no!", 9001);
} catch (error) {
    console.log(`Error value: ${(error as ValueError).value}`);
// Ok

    error.value;
    // Error: Object is of type 'unknown'.
}
```

Non-Null Assertions

Another common use case for type assertions is to remove `null` and/or `defined` from a variable that only theoretically, not practically, might include them. That situation is so common that TypeScript includes a shorthand for it. Instead of writing out `as` and the full type of whatever an object is excluding `null` and `undefined`, you can use a `!` to signify the same thing.

The following two examples are identical in that they both result in `Date` and not `Date | undefined`:

```
// Type: Date | undefined
let maybeDate = Math.random() === 0
    ? undefined
    : new Date();

// Type: Date
maybeDate as Date;

// Type: Date
maybeDate!;
```

Non-null assertions are particularly useful with APIs such as `Map.get` that return a value or `undefined` if it doesn't exist.

Here, `valuesById` is a general `Map`, but we know that it contains a `123` key so the `knownValue` variable can use a `!` to remove `| undefined` from its type.

```
const valuesById = new Map([
    [123, 'temp1'],
    [234, 'temp2'],
]);

// Type: string | undefined
const value = valuesById.get(123);

console.log(value.toUpperCase());
//           ~~~~~
// Error: Object is possibly 'undefined'.

// Type: string
const knownValue = valuesById.get(123)!;

console.log(knownValue.toUpperCase()); // Ok
```

Type Assertion Caveats

Type assertions, like `any`s, are a necessary escape hatch for TypeScript's type system. Therefore, also like `any`, they should be avoided whenever reasonably possible. It is often better to have more accurate types representing your code than it is to make it easier to sneak a technically incorrect object assignment in.

Assertions vs. Declarations

There is a difference between using a type annotation to declare a variable to be of a type verses using a type assertion to change the type of a variable's initial value. TypeScript's type checker performs assignability checking on a variable's initial value against the variable's type annotation when both exist. A type assertion, however, explicitly tells TypeScript to skip some of its type checking.

The following code creates two objects of type `OneTwoTemp` with the same flaw: a missing `two` member. TypeScript is able to catch the error in the `declared` variable because of its `: OneTwoTemp` type annotation. It is not able to catch the error on the `asserted` variable because of the type assertion.

```
interface OneTwoTemp {
    one: string;
    two: number;
}

let declared: OneTwoTemp = {
    one: 'abc',
};
// Error: Property 'two' is missing in type
// '{ one: number; }' but required in type 'OneTwoTemp'.

let asserted = {
    one: 'abc',
} as OneTwoTemp; // Ok, but...

// Both of these statements would fail with:
// TypeError: Cannot read properties of undefined (reading
'toPrecision')
```

```
asserted.two.toPrecision(2);
asserted.two.toPrecision(2);
```

Assertion Assignability

Type assertions are meant to be only a small escape hatch, for situations where the some object's type is just a bit different from reality. TypeScript will only allow type assertions if the new type is either more specific or less specific than the type being destroyed. If the type assertion is between two completely unrelated types then TypeScript will notice and emit a type error.

For example, switching from one primitive to another is not allowed, as primitives have nothing to do with each other:

```
let myValue = '123' as number;
//            ~~~~~~~~~~~~~~~
// Error: Conversion of type 'string' to type 'number'
// may be a mistake because neither type sufficiently
// overlaps with the other. If this was intentional, convert the
expression to 'unknown' first.
```

If you absolutely must switch a value from one type to a totally unrelated type *(why?)*, you can use a double type assertion cast. First cast the value to a top type -`any` or `unknown`- and then cast that result to the unrelated type.

```
let myValueDouble = '123' as unknown as number; // Ok, but...
eww.
```

WARNING

`as unknown as ...` double type assertions are dangerous and almost always a sign of something incorrect in the types of the surrounding code. I teach it here only to help explain the type system and as a precautionary tale, not to encourage its use.

Const Assertions

Back in Chapter 6: Arrays, I introduced an `as const` syntax for changing a mutable array type to a readonly tuple type and promised to use it more later in the book. That time is now!

Const assertions can generally be used to indicate that any value -array, primitive, object, you name it- should be treated as the constant, immutable version of itself. Specifically, `as const` applies the following three rules to whatever type it receives:

- Arrays are treated as `readonly` tuples, not mutable arrays
- Literals are treated as literals, not their general primitive equivalents
- Properties on objects are considered `readonly`

You already saw arrays becoming tuples before, like with this array being asserted as a tuple:

```
// Type: (number | string)[]
[0, ''];

// Type: readonly [0, '']
[0, ''] as const;
```

Let's dig into the other two changes `as const` produces.

Literals to Primitives

It can be useful for the type system to understand a literal value to be that specific literal, rather than widening it to its general primitive.

For example, similar to functions that return tuples, it might be useful for a function to be known to produce a specific literal instead of a general primitive. These functions also return values that can be made more specific — here, `getNameConst`'s return type is the more specific `"Todo"` instead of the general `string`:

```
// Type: () => string
const getName = () => "Todo";

// Type: () => "Todo"
const getNameConst = () => "Todo" as const;
```

It may also be useful to have specific fields on an object be more specific literals. Many popular libraries ask that a discriminant field on an object be a specific literal so the types of their code can more specifically make inferences on the object. Here, the `logAction` variable has a `payload` of type `"log"` instead of `string`, so it can be provided in a location that needs type `LogAction`.

```
interface LogAction {
    payload: string[];
    type: "log";
}

function handleLogAction(action: LogAction) {
    console.log(...action.payload);
}

// Type: { payload: string[], type: "log" }
const logAction = {
    payload: ["Hello", "world!"],
    type: "log" as const,
};

handleLogAction(logAction); // Ok

// Type: { payload: string[], type: string }
const anyAction = {
    payload: ["Hello", "world!"],
    type: "log",
};

handleLogAction(anyAction);
// Error: Argument of type '{ payload: string[]; type: string; }'
// is not assignable to parameter of type 'LogAction'.
//   Types of property 'type' are incompatible.
//     Type 'string' is not assignable to type '"log"'.
```

Readonly Objects

Object literals such as those used as the initial value of a variable generally widen the types of properties to their more general forms the same way the initial values of `let` variables would have. String values such as `'apple'` become `string`, arrays are kept as arrays instead of tuples, and so on. This can be inconvenient when some or all of those values are meant to later be used in a place that requires their specific literal type.

Asserting an object literal with `as const`, however, switches the inferred type to be as specific as possible. All member properties become `readonly`, literals are considered their own literal type instead of their general primitive type, and arrays become readonly tuples. In other words, applying a const assertion to an object literal makes that object literal immutable and recursively applies the same const assertion logic to all its members.

As an example, the `favoritesMutable` object below is declared without an `as const`, so its names are the primitive type `string` and it's allowed to be modified. `favoritesConst`, however, is declared with an `as const`, so its member values are literals and not allowed to be modified.

```
function sayFavorite(fruit: 'apple' | 'banana') {
    console.log(fruit);
}

// Type: { name1: string, name2: string }
const favoritesMutable = {
    name1: 'apple',
    name2: 'banana',
};

sayFavorite(favoritesMutable.name1);
//          ~~~~~~~~~~~~~~~~~~~~~~
// Error: Argument of type 'string' is not assignable
// to parameter of type '"apple" | "banana"'.

favoritesMutable.name1 = 'zucchini'; // Ok

// Type: const { readonly name1: 'apple', name2: 'banana' }
const favoritesConst = {
    name1: 'apple',
```

```
    name2: 'banana',
} as const;

sayFavorite(favoritesConst.name1); // Ok

favoritesConst.name1 = 'zucchini';
//             ~~~~~
// Error: Cannot assign to 'name1' because it is a read-only
property.
```

Summary

Type modifiers allow us to take existing objects and/or types and turn them into new types. In doing so we unlock a hefty amount of extra expressiveness and power in our TypeScript syntax.

- Top types: the highly permissive `any` and the highly restrictive `unknown`
- Type operators: using `keyof` to grab the keys of an type and/or `typeof` to grab the type of an object
- Using -and when not to use- type assertions to sneakily change the type of an object
- Narrowing types using `as const` assertions

Thus concludes the *Features* section of this book. Congratulations: you now know all the most important syntax and type checking features in the TypeScript type system for most projects!

The next section, *Usage*, covers how to configure TypeScript to run on your project, interact with external dependencies, and tweak its type checking and emitted JavaScript. Those are important features for using TypeScript on your own projects.

There are some other miscellaneous type operations available in TypeScript syntax. You don't need to fully understand them to work in most TypeScript projects — but they are interesting and useful to know. I've thrown them in an *Extra Credit* section after the end of the *Usage* section as a fun little treat if you have the time.

Part III. Usage

Chapter 10. Declaration Files

What do TypeScript types

share with southern USA?

“Well, they do `declare`!”

As great as writing code in TypeScript is, TypeScript projects frequently need to be able to work with raw JavaScript files. Many packages are written directly in JavaScript, not TypeScript. Even packages that are written in TypeScript distribute built code as `.js` files.

Moreover, TypeScript projects need a way to be told the type shapes of environment-specific features such as global variables and APIs. A project running in, say, Node.js might have access to built-in Node modules not available in browsers — and vice versa.

TypeScript allows declaring type shapes separately from their implementation using files with the `.d.ts` definition, known as *declaration files*. Declaration files are generally either written within a project, built and distributed with a project's compiled npm package, or shared as a standalone "typings" package.

Declaration Files

A `.d.ts` declaration file generally works similarly to a `.ts` file, except with the notable constraint of not being allowed to include runtime code. `.d.ts` files contain only descriptions of interfaces, modules, and general types. They don't contain any runtime code that could be compiled down to JavaScript.

Developers sometimes use `.d.ts` to store list of types in an application. They can be imported from just like any other source TypeScript file.

This `types.d.ts` file exports a `FruitTemp` interface used by an `index.ts` file:

```
// types.d.ts
export interface FruitTemp {
    juicy: boolean;
    name: string;
}

// fruit.ts
import { FruitTemp } from "./types";

export const myFruit: FruitTemp = {
    juicy: true,
    name: "temp",
};
```

TIP

Declaration files create what's known as an *ambient context*: meaning types without an implementation. Error messages that refer to ambient contexts are often a result of trying to add an implementation to a type shape where not allowed.

Declaring Runtime Values

Although definition files may not create runtime values such as functions or variables, they are able to declare that those constructs exist with the `declare` keyword. Doing so tells the type system that some external influence has created the value under that name with a particular type shape.

Declaring a variable with `declare` uses the same syntax as a normal variable declaration, except an initial value is not allowed. This snippet successfully declares a variable `temp1` but receives a type error for trying to give a value to `temp2`:

```
declare let temp1: string; // Ok

declare let temp2: string = "value";
//                          ~~~~~~~
// Error: Initializers are not allowed in ambient contexts.
```

Functions and classes are also declared similarly to their normal forms, but without the bodies of functions or methods.

```
function temp1(): number; // Ok

function temp2() { return 0; }
//                ~
// Error: An implementation cannot be declared in ambient
contexts.

class Temp3 {
    temp4(): number; // Ok

    temp5() {
        //  ~
        // Error: An implementation cannot be declared in ambient
contexts.
        return 0;
    }
}
```

TIP

TypeScript's implicit `any` rules work the same for functions and variables declared in ambient contexts as they do in normal source code. Because ambient contexts may not provide function bodies or initial variable values, explicit type annotations are generally the only way to stop them from implicitly being type `any`.

While type shapes such as interfaces are allowed with or without a `declare`, runtime constructs such as functions or variables will trigger a type complaint without a `declare`:

```
interface Temp1 {} // Ok
declare interface Temp2 {} // Ok

declare const temp3: string; // Ok: temp3 is the primitive string
type
declare const temp4: "value"; // Ok: temp4 is the literal "value"
type

const temp3 = "value";
// Error: Top-level declarations in .d.ts files must
// start with either a 'declare' or 'export' modifier.
```

Global Values

TODO: MODULES HAVE NOT BEEN EXPLAINED IN A PREVIOUS CHAPTER! I'm thinking an early one about type checking, maybe?

Because TypeScript files that have no `import` or `export` statements are treated as *scripts* rather than *modules*, constructs -including types- declared in them are available globally. Definition files without any imports or exports can take advantage of that behavior to declare types globally. Global definition files are particularly useful for declaring global types or variables available across all files in an application.

Here, a `globals.d.ts` file declares that a `const version: string` exists globally. A `version.ts` file is then able to refer to a global `version` variable despite not importing from `globals.d.ts`:

```
// globals.d.ts
declare const version: string;

// version.ts
export function logVersion() {
    console.log(`Version: ${version}`); // Ok
}
```

Globally declared values are most often used in browser applications that use global variables. Although most modern web frameworks use newer techniques such as ECMAScript modules, it can still be useful -especially in smaller projects- to be able to store variables globally.

Global Interface Merging

Variables aren't the only globals floating around in a TypeScript project's type system. Many type declarations exist globally for global APIs and values. Interfaces declared globally may merge with interfaces of the same name declared in global script files — most commonly `.d.ts` declaration files.

For example, a web application that relies on a global variable set by the server might want to declare that as existing on the global `Window` interface. Interface merging would allow a file such as `types/window.d.ts` to declare a variable that exists on the global `window` variable of type `Window`:

```
// types/window.d.ts
interface Window {
    myVersion: string;
}

// index.ts
export function logWindowVersion() {
    console.log(`Window version is: ${window.myVersion}`);
}
```

Global Augmentations

It’s not always feasible to refrain from `import` or `export` statements in a `.d.ts` file that needs to also augment the global scope. Sometimes types declared in a module file are meant to be consumed globally.

For those cases, TypeScript allows a syntax to `declare global` a block of code. Doing so marks the contents of that block as being in a global context even though their surroundings are not.

```
// (module context)

declare global {
    // (global context)
}

// (module context)
```

Here, a `types/data.d.ts` file exports a `DataTemp` interface imported by both `types/globals.d.ts` and the runtime `index.ts`. `types/globals.d.ts` declares a variable of type `DataTemp` globally inside a `declare global` block as well as a variable available only in that file.

```
// types/data.d.ts
export interface DataTemp {
    version: string;
}

// types/globals.d.ts
import { DataTemp } from './data';

declare global {
    const globallyDeclared: DataTemp;
}

declare const locallyDeclared: DataTemp;

// index.ts
import { DataTemp } from './types/data';

function logDataTemp(data: DataTemp) { // Ok
    console.log(`Data version is: ${data.version}`);
}

logDataTemp(globallyDeclared); // Ok

logDataTemp(locallyDeclared);
//          ~~~~~~~~~~~~~~~
// Error: Cannot find name 'locallyDeclared'.
```

Built-In Declarations

Now that you've seen how declarations work, it's time to unveil their hidden use in TypeScript: they've been powering its type checking the whole time! Global objects such as `Array`, `Function`, `Map`, and `Set` are examples of constructs that the type system needs to know about but aren't declared in your code. They're provided by whatever runtime(s) your code is meant to run in: Deno, Node, a web browser, etc.

Library Declarations

To start, the built-in global objects such as `Array` and `Function` that exist in all JavaScript runtimes are declared in a file with names like `lib.[target].d.ts`. `target` is the minimum support version of JavaScript targeted by your project, such as `es5`, `es2020`, or `esnext`.

The built-in library definition files, or "lib files", are fairly large because they represent the entirety of JavaScript's built-in APIs. For example, methods on the built-in `Number` type are represented by a global `var Number: NumberConstructor` interface that starts like:

```
// lib.es5.d.ts

/**
 * An object that represents a number of any kind.
 * All JavaScript numbers are 64-bit floating-point numbers.
 */
declare var Number: NumberConstructor;

interface NumberConstructor {
    new(value?: any): Number;
```

```
    (value?: any): number;
    readonly prototype: Number;

    /**
     * The largest number that can be represented in JavaScript.
     * Equal to approximately 1.79E+308.
     */
    readonly MAX_VALUE: number;

    // ...
}
```

Lib files are distributed as part of the TypeScript npm package. You can find them inside the package at paths like `node_modules/typescript/lib/lib.es5.d.ts`. For IDEs such as VS Code that use their own packaged TypeScript versions to type check code, you can find the lib file being used by right-clicking on a name such as `Number` in your code and selecting an option like *Go to Type Definition*.

TODO SCREENSHOT

Library Targets

TypeScript by default will include the appropriate lib file based on the `target` setting provided to the `tsc` CLI and/or in your project's `tsconfig.json` (by default, `es5`). Successive lib files for newer versions of JavaScript build on each other using interface merging.

For example, the static `Number` members such as `EPSILON` and `isFinite` added in ES2015 are listed in `lib.es2015.d.ts`:

```
// lib.es2015.d.ts

interface NumberConstructor {
    /**
     * The value of Number.EPSILON is the difference between 1
and the
     * smallest value greater than 1 that is representable as a
Number
     * value, which is approximately:
     * 2.2204460492503130808472633361816 x 10−16.
     */
    readonly EPSILON: number;

    /**
     * Returns true if passed value is finite.
     * Unlike the global isFinite, Number.isFinite doesn't
forcibly
     * convert the parameter to a number. Only finite values of
the
     * type number result in true.
     * @param number A numeric value.
     */
    isFinite(number: unknown): boolean;

    // ...
}
```

TypeScript projects will include the lib files for all version targets of JavaScript up through their minimum target. For example, a project with a `target` of `es2016` would include `lib.es5.d.ts`, `lib.es2015.d.ts`, and `lib.es2016.d.ts`.

TIP

Language features available only in newer versions of JavaScript than your target will not be available in the type system. For example, if your `target` is `es5`, language features from `es2015` or later such as `String.prototype.startsWith` will not exist.

Compiler options such as `target` are covered in more detail in Chapter 13: Configuration Options.

DOM Declarations

Outside of the JavaScript language itself, the most commonly referenced area of type declaration is for the web browsers. Web browser types, generally referred to as “DOM”, cover APIs such as `localStorage` and type shapes such as `HTMLElement` available only in web browsers. DOM types are stored in a `lib.dom.d.ts` file alongside the other `lib.*.d.ts` declaration files.

Global DOM types, like built-in globals, are generally described with global interfaces. For example, the `Storage` interface is used for `localStorage` and `sessionStorage` and starts roughly like:

```
// lib.dom.d.ts

interface Storage {
    /**
     * Returns the number of key/value pairs.
     */
    readonly length: number;

    /**
     * Removes all key/value pairs, if there are any.
     */
    clear(): void;

    /**
     * Returns the current value associated with the given key,
     * or null if the given key does not exist.
     */
    getItem(key: string): string | null;

    // ...
}
```

TypeScript includes DOM types by default in projects that don't override the `lib` compiler option. That can sometimes be confusing for developers working on projects meant to be run in non-browser environments such as Node, as they shouldn't be able to access the global APIs such as `document` and `localStorage` that the type system would then claim to exist. Compiler options such as `lib` are covered in more detail in Chapter 13: Configuration Options.

Module Declarations

One more important feature of declaration files is their ability to describe the shapes of modules. The `declare` keyword can be used before a string name of a module to inform the type system of the contents of that module.

Here, a `"temp"` module is declared as being in existence in a `modules.d.ts` declaration script file, then used in an `index.ts` file:

```
// modules.d.ts
declare module "temp" {
    export function logStuff(): void;
}

// index.ts
import { logStuff } from "temp";

logStuff();
```

Wildcard Module Declarations

A common use of module declarations is to tell web applications that a particular non-JavaScript/TypeScript file extension is available to `import` into code. Module declarations may contain a single `*` wildcard to indicate that any module matching that pattern looks the same.

For example, many web projects use CSS modules to `import styles` from `.scss` files as objects that can be used at runtime. They would define a `"*.css` module that default exports an object of type `Record<string, string>`:

```
// styles.d.ts
declare module "*.css" {
    const styles: Record<string, string>;
    export default styles;
}

// component.ts
import styles from "./component-styles.css";

styles.anyClassName; // Type: string
```

WARNING

Using wildcard modules to represent local files isn't completely type safe. TypeScript does not provide a mechanism to ensure the imported module path matches a local file. Most projects use a build system such as Webpack and/or generate `.d.ts` files from local files to make sure imports match up.

Package Types

Now that you've seen how to declare typings within a project, it's time to cover consuming types between packages. Project written in TypeScript still generally distribute packages containing compiled `.js` outputs. They typically use `.d.ts` files to declare the backing TypeScript type system shapes behind those JavaScript files.

declaration

TypeScript provides a `declaration` option to create `.d.ts` outputs for input files alongside JavaScript outputs.

For example, given the following `index.ts` source file:

```
// index.ts
export const greet = (text: string) => {
    console.log(`Hello, ${text}!`);
};
```

Using `declaration`, a `module` of `es2015`, and a `target` of `es2015`, the following outputs would be generated:

```
// index.d.ts
export declare const greet: (text: string) => void;

// index.js
export const greet = (text) => {
    console.log(`Hello, ${text}!`);
};
```

Auto-generated `.d.ts` files are the best way for a project to create type definitions to be used by consumers. It's generally recommended that most packages written in TypeScript that produce `.js` file outputs should also bundle `.d.ts` alongside those files.

Compiler options such as `declaration` are covered in more detail in Chapter 13: Configuration Options.

Package Types

TypeScript is able to automatically detect and utilize `.d.ts` files bundled inside a project's `node_modules` dependencies. Those files will inform the type system about the type shapes exported by that package as if they were written inside the same project or declared with a `declare` module block.

A typical npm module that comes with its own `.d.ts` declaration files might have a file structure something like:

```
lib/
    index.js
    index.d.ts
package.json
```

As an example, the ever-popular test runner Jest is written in TypeScript and provides its own bundled `.d.ts` files in its `jest` package. It has a dependency on the `@jest/globals` package that provides functions such as `describe` and `it`, which `jest` then makes available globally:

```
// package.json
{
    "devDependencies": {
        "jest": "^32.1.0"
    }
}

// using-globals.d.ts
describe("MyAPI", () => {
    it("works", () => { /* ... */ });
});

// using-imported.d.ts
import { describe, it } from "@jest/globals";

describe("MyAPI", () => {
    it("works", () => { /* ... */ });
});
```

If we were to recreate a very limited subset of the Jest typings packages from scratch, they might look some something like these files:

```
// node_modules/jest/index.d.ts
import * as globals from '@jest/globals';

declare global {
    const describe: typeof globals.describe;
    const it: typeof globals.it;
}

// node_modules/@jest/globals/index.d.ts
export function describe(name: string, test: () => void);
export function it(name: string, test: () => void);
```

DefinitelyTyped

Sadly, not all projects are written in TypeScript. Some unfortunate developers are still writing their projects in plain old JavaScript without a type checker to aide them. Horrifying.

Our TypeScript projects still need to be informed of the type shapes of the modules from those packages. The TypeScript team and community created a giant repository called DefinitelyTyped to house community-authored definitions for packages. DefinitelyTyped, or DT for short, is one of the most active repositories on GitHub and contains thousands of packages of `.d.ts` definitions, along with automation around reviewing change proposals and publishing updates.

DT packages are published on npm under the `@types` scope with the same name as the package they provide types for. `@types/react`, for example, provides type definitions for the `react` package as of 2022.

> **NOTE**
>
> `@types` are generally installed as either `dependencies` or `devDependencies`, though the distinction between those two has become blurred in recent years. In general, if your project is meant to be distributed as an npm package, it should use `dependencies` so that consumers of the package also bring in the type definitions used within. If your project is a standalone application such as one built and run on a server, it should use `devDependencies` to convey that the types are just a development-time tool.

For example, for a utility package that relies on `lodash` -which as of 2022 has a separate `@types/lodash` package-, the `package.json` would contain lines similar to:

```
// package.json
{
    "dependencies": {
        "@types/lodash": "^4.14.178",
        "lodash": "^4.17.21",
    }
}
```

The `package.json` for a standalone app built on React might contain lines similar to:

```
// package.json
{
    "dependencies": {
        "react": "^17.0.2"
    },
    "devDependencies": {
        "@types/react": "^17.0.38"
    },
}
```

WARNING

Semantic versioning (“semver”) numbers do not necessary match between `@types/` packages and the packages they represent. You may often find some that are off by a patch version as with React earlier, a minor version as with Lodash earlier, or even major versions.

TIP

See ***https://aka.ms/types*** for a handy search tool to display whether a package has types bundled or via a separate `@types/` package.

Chapter 11. Configuration Options

Compiler options:

Types and modules and oh my!

`tsc` *your way.*

TypeScript is highly configurable and made to adapt to all common JavaScript usage patterns. It can work for projects ranging from legacy browser code to the most modern server environments.

Much of TypeScript's configurability comes from its cornucopia of over 100 configuration options that can be provided via either of:

- Command-line (CLI) flags passed to `tsc`
- "TSConfig" TypeScript configuration files

This chapter is not intended as a full reference for all TypeScript configuration options. Instead, I've included just the ones that tend to be more useful and widely used for most TypeScript project setups. See **aka.ms/tsc** for a full reference on each of these options and more.

`tsc` options

Back in Chapter 1, you used `tsc index.ts` to compile an `index.ts` file. The `tsc` command can take in most of TypeScript's configuration options as `--` flags.

For example, to run `tsc` on an `index.ts` file and skip emitting an `index.js` file (so, only run type checking), pass the `--noEmit` flag:

```
tsc index.ts --noEmit
```

You can run `tsc --help` to get a list of commonly used CLI flags. The full list of `tsc` configuration options is viewable with `tsc --all` and documented on *https://aka.ms/tsc-reference*.

Pretty Mode

The `tsc` CLI has the ability to output in a "pretty" mode: stylized with colors and spacing to make them easier to read. It defaults to pretty mode if it detects that the output terminal supports colorful text.

Here's an example of what `tsc` looks like printing two type errors from a file:

```
# TEMP: WILL REPLACE WITH SCREENSHOT THAT INCLUDES COLORS
# tsc index.ts
index.ts:1:12 - error TS2322: Type 'string' is not assignable to
type 'number'.

1 export let notNumeric: number = "Gotcha!";
             ~~~~~~~~~~

index.ts:2:12 - error TS2322: Type 'number' is not assignable to
```

```
type 'string'.

2 export let notString: string = 1337;
             ~~~~~~~~~

Found 2 errors.
```

If you'd prefer CLI output that is more condensed and/or doesn't have different colors, you can explicitly provide `--pretty false` to tell TypeScript to use a more terse, uncolored format:

```
# TEMP: WILL REPLACE WITH SCREENSHOT THAT INCLUDES COLORS BEFORE
PUBLISHING
# tsc --pretty false index.ts
index.ts(1,12): error TS2322: Type 'string' is not assignable to
type 'number'.
index.ts(2,12): error TS2322: Type 'number' is not assignable to
type 'string'.
```

Watch Mode

My favorite way to use the `tsc` CLI is with its `-w`/`--watch` mode. Instead of exiting once completed, watch mode has it stay running indefinitely and continuously update your terminal with a real-time list of all the errors it sees.

Running in watch mode on a file that contains an error:

```
# TEMP: WILL REPLACE WITH SCREENSHOT THAT INCLUDES COLORS BEFORE
PUBLISHING
[7:06:59 PM] Starting compilation in watch mode...

index.ts:1:12 - error TS2322: Type 'number' is not assignable to
type 'string'.

1 export let notString: string = 1337;
             ~~~~~~~~~

[7:07:00 PM] Found 1 error. Watching for file changes.
```

After fixing the error and saving the file:

```
[7:07:42 PM] File change detected. Starting incremental
compilation...

[7:07:42 PM] Found 0 errors. Watching for file changes.
```

Watch mode is particularly useful when you're working on large changes such as refactors across many files. You can use TypeScript's type errors as a checklist-of-sorts to see what still needs to be cleaned up.

TSConfig Files

Alternately, instead of always providing all file names and configuration options to `tsc`, most configuration options may be specified

A `tsconfig.json` ("TSConfig") file in a directory indicates that the directory is the root of a TypeScript project. Running `tsc` in a directory will read in any configuration options in that `tsconfig.json` file.

You can also pass `-p`/`--project` to `tsc` with a path to a directory containing a `tsconfig.json` or any file to have `tsc` use that instead:

```
tsc -p path/to/tsconfig.json
```

NOTE

TypeScript approximately never changes the default TypeScript configuration options, to maintain compatibility between versions.

`tsc --init`

The `tsc` command-line includes an `--init` command to create a new `tsconfig.json` file. That newly created TSConfig file will contain a link to the configuration docs as well as most of the allowed TypeScript configuration options with one-line comments briefly describing their use.

```
tsc --init
```

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig.json to read more about this
file */
    // ...
  }
}
```

CLI vs. Configuration

Looking through the TSConfig file created by `tsc --init`, you may notice that configuration options in that file are within a `"compilerOptions"` object. Most options available in both the CLI and in TSConfig files are generally split into two categories:

- **Compiler**: how each included file is compiled and/or type-checked by TypeScript
- **File**: which files will or will not have TypeScript run on them

Other settings covered below, such as project references, generally are only available in TSConfig files.

TIP

If a setting is provided to the `tsc` CLI, it will generally override any value specified in a TSConfig file. Because IDEs generally read from the `tsconfig.json` in a directory for TypeScript settings, it's recommended to put most configuration options in a `tsconfig.json` file.

File Inclusions

By default, `tsc` will run on all non-hidden files (those whose name does not start with a `.`) in the current directory and any child directories, ignoring directories named `.git` or `node_modules`. TypeScript configurations can change that list of files to run on.

`files`

The simplest way to tell TypeScript to run on a different list of files is to provide it a value for its `files` configuration option. The `tsc` CLI can take in a specific list of file names to run on, or it can be specified as a top-level `"files"` option in a TSConfig file.

This CLI usage and the following configuration file are equivalent in directing TypeScript to run on files `a.ts`, `b.ts`, and `c.ts`:

```
tsc a.ts b.ts c.ts
```

```
{
  "files": ["a.ts", "b.ts", "c.ts"]
}
```

Note that any file imported by a source file included in a `files` list will be added to the list of files TypeScript will run, regardless of whether it was explicitly listed in `files`.

NOTE

The `files` configuration option does not support any form of glob wildcards. Most developers typically choose the following, more flexible `include` configuration option.

`include`

The most common way to include files is with a top-level `"include"` property in a `tsconfig.json`. It allows an array of strings that describe what directories and/or files to include in TypeScript compilation.

For example, this configuration file includes all TypeScript source files in a `src/` directory relative to the `tsconfig.json`:

```
{
  "include": ["src"]
}
```

Glob wildcards are allowed in `include` strings for more fine-grained control of files to include.

- `*` matches zero or more characters (excluding directory separators)
- `?` matches any one character (excluding directory separators)
- `**/` matches any directory nested to any levels

This configuration file allows only `.d.ts` files nested in a `typings/` directory and `src/`` files with at least two characters in their name before an extension:

```json
{
  "include": [
    "typings/**/*.d.ts",
    "src/**/*??.*"
  ]
}
```

`exclude`

The `include` list of files for a project sometimes includes files not meant for compilation by TypeScript. TypeScript allows a TSConfig file to omit paths from `include` by specifying them in a top-level `"exclude"` property. Similar to `include`, it allows an array of strings that describe what directories and/or files to exclude from TypeScript compilation.

The following configuration includes all files in `src/` except for those within any nested `external/` directory and a `node_modules` directory:

```json
{
  "exclude": ["**/external", "node_modules"],
  "include": ["src"]
}
```

By default, `exclude` contains `["node_modules", "bower_components", "jspm_packages"]` to avoid running the TypeScript compiler on compiled third-party library files.

> **TIP**
>
> Unless your project happens to use Bower or JSPM, only `"node_modules"` typically needs to be re-added from the default `exclude` list.

Syntax Extensions

TypeScript is by default able to read in any file whose extension is `.ts`. However, some projects require being able to read in files with different extensions, such as JSON modules or JSX syntax for UI libraries such as React.

JSX Syntax

JSX syntax like `<Component />` is often used in UI libraries such as Preact and React. JSX syntax is not technically JavaScript. Like TypeScript's type definitions, it's an extension to JavaScript syntax that compiles down to regular JavaScript.

```
const MyComponent = () => {
  return (
    <div>Hello, world!</div>
  )
}
```

In order to use JSX syntax in a file, you must do two things:

- Name that file with a `.tsx` extension
- Enable the `"jsx"`` compiler option in your configuration options

`jsx`

The value used for the `"jsx"` compiler option determines how TypeScript emits JavaScript code for `.tsx` files. Projects generally use one of these three values:

Value	Input Code	Output Code	Output File Extension
“preserve”	`<div />`	`<div />`	.jsx
“react”	`<div />`	`React.createElement("div")`	.js
“react-native”	`<div />`	`<div />`	.js

Values for `jsx` may be provided to the `tsc` CLI and/or in a TSConfig file:

```
tsc --jsx preserve
```

```
{
  "compilerOptions": {
    "jsx": "preserve"
  }
}
```

If you’re not using TypeScript’s built-in transpiler, you can use any of the values. That includes web apps built on some kind of framework such as create-react-app or Next.js that handle React configuration and compiling syntax.

`resolveJsonModule`

TypeScript will allow reading in `.json` files if the `resolveJsonModule` compiler option is set to `true`. When it is, `.json` files may be imported from as if they were `.ts` files.

For JSON files that contain an object, destructuring imports may be used. This pair of files defines an `"author"` string in a `author.json` file and imports it into a `usesAuthor.ts` file:

```
// author.json
{
  "author": "Mary Astell"
}

// usesAuthor.ts
import { author } from "./author.json";

// Logs: "Mary Astell"
console.log(author);
```

For JSON files that contain other literal types, such as arrays or numbers, you'll have to use the `* as` import syntax. This pair of files defines an array of strings in an `authors.json` that is then imported into a `usesAuthors.ts` file:`

```
// authors.json
[
    "Damaris Cudworth Masham",
    "Vita Sackville-West",
    "Virginia Woolf"
]

// usesAuthors.ts
import * as authors from "./authors.json";

// Logs: "3 authors"
console.log(`${authors.length} authors`);
```

Emit

Although the rise of dedicated compiler tools such as Babel and SWC has reduced TypeScript's role in some projects to solely type checking, many other projects still rely on TypeScript for compiling TypeScript syntax to JavaScript. It's quite useful for projects to be able to take in a single dependency on `typescript` and use its `tsc` command to output the equivalent JavaScript.

`outDir`

By default, TypeScript places output files alongside their corresponding source files. For example, running `tsc` on a directory containing `src/fruits/apple.ts` and `src/vegetables/zucchini.ts` would result with a structure like:

```
fruits/
  apple.js
  apple.ts
vegetables/
  zucchini.js
  zucchini.ts
```

Sometimes it may be preferable to place output files in a different folder. Many Node projects, for example, standardize on putting transformed outputs in a `dist` or `lib` directory.

TypeScript's `outDir` compiler option allows specifying a different root directory for outputs. Output files are kept in the same relative directory structure as input files.

For example, running `tsc --outDir dist` on the above directory would place outputs within a `dist/` folder:

```
dist/
  fruits/
    apple.js
  vegetables/
    zucchini.js
fruits/
  apple.ts
vegetables/
  zucchini.ts
```

TypeScript calculates the root directory to place output files into by finding the longest common sub-path of all input files (excluding `.d.ts` declaration files). That means that projects that place all input source files in a single directory will have that directory treated as the root.

For example, if the above example put all inputs in a `src/` directory and compiled with `--outDir lib`, `lib/fruits/apple.js` would be created instead of `lib/src/fruits/apple.js`:

```
lib/
  fruits/
    apple.js
  vegetables/
    zucchini.js
src/
  fruits/
    apple.ts
  vegetables/
    zucchini.ts
```

A `rootDir` compiler option does exist to explicitly customize that root directory, but it’s rarely necessary or used with values other than `.` or `src`.

target

TypeScript is able to produce output JavaScript that can run in environments as old as ES3 (circa 1999!). Most environments are able to support syntax features from much newer versions of JavaScript.

TypeScript includes a `target` compiler option to specify how far back in syntax support JavaScript code needs to be transpiled. Although `target` defaults to `es3` for backwards compatibility reasons, it’s generally advisable use as new JavaScript syntax as possible per your target platform(s), as supporting newer JavaScript features in older environments necessitates creating more JavaScript code.

TIP

As of 2022, all releases within the last year of browsers serving >0.1% of worldwide users support at least all of ECMAScript 2019 and nearly all of ECMAScript 2020-2021, while the LTS supported versions of Node.js support all of ECMAScript 2022. There's very little reason not to have a `target` at least as high as `es2019`.

For example, given this TypeScript source containing ES2015 `const`s and ES2020 `??` nullish coalescing:

```
function defaultNameAndLog(nameMaybe: string | undefined) {
  const name = nameMaybe ?? "anonymous";
  console.log("From", nameMaybe, "to", name);
  return name;
}
```

With `tsc --target es2020` or newer, both `const` and `??` are supported syntax features, so TypeScript would only need to remove the `: string | undefined` from that snippet:

```
function defaultNameAndLog(nameMaybe) {
  const name = nameMaybe ?? "anonymous";
  console.log("From", nameMaybe, "to", name);
  return name;
}
```

With `tsc --target es5` or `tsc --target es2015` through `es2019`, the `??` syntax sugar would be compiled down to its equivalent in the older versions of JavaScript:

```
function defaultNameAndLog(nameMaybe) {
    const name = nameMaybe !== null && nameMaybe !== void 0 ?
nameMaybe : "anonymous";
    console.log("From", nameMaybe, "to", name);
    return name;
}
```

With `tsc --target es3` or `es5`, the `const` would additionally need to be converted to its equivalent `var`:

```
function defaultNameAndLog(nameMaybe) {
    var name = nameMaybe !== null && nameMaybe !== void 0 ?
nameMaybe : "anonymous";
    console.log("From", nameMaybe, "to", name);
    return name;
}
```

> **TIP**
>
> Think of the `lib` compiler option as indicating what type system features are available, whereas the `target` compiler option as indicating what syntax features exist.

Emitting Declarations

TODO: WILL FILL THIS OUT AFTER CHAPTER 11 COVERS THEM (IT IS NOT YET CLEAR WHAT WILL NEED TO BE COVERED)

emitDeclarationOnly

As a specialized alternative to the `declaration` compiler option, an `emitDeclarationOnly` compiler option exists that directs TypeScript to only emit declaration files: no `.js`/`.jsx` files at all. This can be useful for projects that use an external tool to generate output JavaScript but still want to use TypeScript to generate output definition files.

```
tsc --emitDeclarationOnly
```

```
{
  "compilerOptions": {
    "emitDeclarationOnly": true
  }
}
```

Source Maps

TypeScript includes the ability to output source map alongside output files. Source maps are descriptions of how transformed output files such as `.js` files map back to their original sources such as `.ts` files. They allow developer tools such as debuggers to display original source code when navigating through the output file.

`sourceMap`

TypeScript's `sourceMaps` compiler option enables outputting `.js.map` or `.jsx.map` sourcemaps alongside `.js` or `.jsx` output files. Source map files are otherwise given the same name as their corresponding output JavaScript file and placed in the same directory. For example, compiling with `tsc src/index.ts --sourceMap` would produce `src/index.js` and `src/index.js.map`.

`declarationMap`

TypeScript is also able to generate source maps for `.d.ts` declaration files. Its `declarationMap` compiler option directs it to generate a `.d.ts.map` source map for each `.d.ts` that maps back to the original source file. Declaration maps enable IDEs such as VS Code to go to the original source file when using editor features such as Go to Definition.

TIP

`declarationMap` is particularly useful when working with project references, covered towards the end of this chapter.

`noEmit`

For projects that completely rely on other tools to compile source files to output JavaScript, TypeScript can be told to skip emitting files altogether. Enabling the `noEmit` compiler option directs TypeScript to act purely as a type checker.

Type Checking

TypeScript’s type checker is given the most amount of configuration options of any of TypeScript’s areas of features. You can configure it to be gentle and forgiving, only emitting type checking complaints when it’s completely certain of an error, or harsh and strict, requiring all code be well-typed.

`lib`

To start, which global APIs TypeScript assumes to be present in the runtime environment is configurable with the `lib` compiler option. It takes in an array of strings that defaults to your `target` compiler option, as well as `dom` to indicate including browser types.

Most of the time, the only reason to customize `lib` would be to remove the `dom` inclusion for a project that doesn’t run in the browser:

```
tsc --lib es2020
```

```
{
  "compilerOptions": {
    "lib": ["es2020"]
  }
}
```

Alternately, for a project that uses polyfills to support newer JavaScript APIs in a browser, `lib` can include `dom` and any ECMAScript version:

```
tsc --lib dom,es2020
```

```
{
  "compilerOptions": {
    "lib": ["dom", "es2021"]
  }
}
```

TODO SPECIFICS (COMMONLY POLYFILLED) SUCH AS ITERABLE

`skipLibCheck`

TypeScript provides a `skipLibCheck` compiler option that indicates to skip type checking in declaration files not explicitly included in your source code. This can be useful for applications that rely on many dependencies which may rely on different, conflicting definitions of shared libraries.

```
tsc --skipLibCheck
```

```json
{
  "compilerOptions": {
    "skipLibCheck": true
  }
}
```

Strict Mode

Most of TypeScript's type checking compiler options are grouped into what TypeScript refers to as *strict mode*. Each strictness compiler option defaults to `false`, and when enabled, directs the type checker to turn on some addition checks.

You can enable all strict mode checks by enabling the `strict` compiler option:

```
tsc --strict
```

```json
{
  "compilerOptions": {
    "strict": true
  }
}
```

If you want to enable all strict mode checks except for certain ones, you can both enable `strict` and explicitly disable certain checks. For example, this configuration enables all strict modes except for `noImplicitAny`:

```
tsc --strict --noImplicitAny false
```

```json
{
  "compilerOptions": {
    "noImplicitAny": false,
    "strict": true
  }
}
```

WARNING

Future versions of TypeScript may introduce new strict type checking compiler options under `strict`.

`noImplicitAny`

If TypeScript cannot infer the type of a parameter or property, then it will fall back to assuming the `any` type. It is generally best practice to not allow `any` types in code as the `any` allows a value to bypass much of the type checker.

The `noImplicitAny` compiler option directs TypeScript to issue a type checking complaint when it has to fall back to an implicit `any`.

For example, writing the following function parameter without a type definition would cause a type error under `noImplicitAny`:

```ts
const logMessage = (message) => {
  //                ~~~~~~~
  // Error: Parameter 'message' implicitly has an 'any' type.
  console.log(`Message: ${message}!`);
};
```

Most of the time, a `noImplicitAny` complaint can be resolved either by adding a type annotation on the complaining location:

```ts
const logMessage = (message: string) => { // Ok
  console.log(`Message: ${message}!`);
}
```

…or, in the case of function parameters, putting the parent function in a location that indicates the type of the function:

```
type LogsMessage = (message: string) => void;

const logMessage: LogsMessage = (message) => { // Ok
  console.log(`Message: ${message}!`);
}
```

TIP

`noImplicitAny` is an excellent flag for ensuring type safety across a project. However, using it while converting an existing application to TypeScript or to block any code from being built during local development can be a pain.

`noImplicitThis`

Similar to parameters and properties with `noImplicitAny`, the `this` reference in `function`s defaults to `any` unless specified. Although most TypeScript projects generally use arrow functions and/or classes to avoid having to use `this`, there are still some times when it's useful to use `this`.

Writing the following function that uses `this` without declaring its type would cause a type error under `noImplicitThis`:

```
export function MessageBox(message: string) {
  this.message = message;
  // Error: 'this' implicitly has type 'any'
  // because it does not have a type annotation.
}
```

`noImplicitThis` complaints that show up in code written before JavaScript ES2015 classes can generally be resolved by converting the code to using classes.

For example, the above function could be converted to a class and constructor:

```
class MessageBox {
  message: string;

  constructor(message: string) {
    this.message = message; // Ok
  }
}
```

If the function truly is a standalone function not associated with a particular class, you can instead create or reuse a type for the `this`:

```
interface HasMessage {
  message: string;
}

function logsMessage(this: Message) {
  console.log("My message is:", this.message); // Ok
}
```

`strictBindCallApply`

When TypeScript was first released, it didn't have rich enough type system features to be able to represent the built-in `Function.apply`, `Function.bind`, or `Function.call`. Those functions by default had to take in `any` for their list of arguments. That's not very type-safe!

As an example, without `strictBindCallApply`, the following variations on `getLength` all include `any` in their types:

```
function getLength(text: string, trim?: boolean) {
  return trim ? text.trim().length : text;
}

// Type: (this: Function, argArray?: any) => any
getLength.apply;

// Type: any
getLength.bind(undefined, "temp");

// Type: any
getLength.call(undefined, "temp", true);
```

Now that TypeScript's type system features are powerful enough to represent those functions' generic rest arguments, TypeScript allows opting in to using more restrictive types for the functions.

Enabling `strictBindCallApply` enables much more precise types for the `getLength` variations:

```
function getLength(text: string, trim?: boolean) {
  return trim ? text.trim().length : text;
}

// Type: (this: typeof getLength, args: [text: string, trim?:
boolean]) => number;
getLength.apply;

// Type: (trim?: boolean) => number
getLength.bind(undefined, "temp");

// Type: number
getLength.call(undefined, "temp", true);
```

`strictFunctionTypes`

The `strictFunctionTypes` compiler causes function type parameters to be checked slightly more strictly. A function type is no longer considered assignable to another function type if its parameters are subtypes of that other type's parameters.

As a concrete example, the `checkOnNumber` function here takes in a function that should be able to receive a `number | string`, but is provided with a `stringContainsA` function that expects to take in a parameter only of type `string`. TypeScript's default bivariant type checking would allow it — and the program would crash from trying to call `.match()` on a `number`!.

```
function checkOnNumber(containsA: (input: number | string) =>
boolean) {
  return containsA(1337);
}

function stringContainsA(input: string) {
    return !!input.match(/a/i);
}

checkOnNumber(stringContainsA);
```

Under `strictFunctionTypes`, the `checkOnNumber(stringContainsA)` would cause a type checking error:

```
// Argument of type '(input: string) => boolean' is not
assignable
// to parameter of type '(input: string | number) => boolean'.
//   Types of parameters 'input' and 'input' are incompatible.
//     Type 'string | number' is not assignable to type 'string'.
//       Type 'number' is not assignable to type 'string'.
checkOnNumber(stringContainsA);
```

NOTE

In technical terms, function parameters switch from being *bivariant* to *contravariant*. You can read more about the difference in the TypeScript 2.6 release notes: *https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-6.html*.

`strictNullChecks`

Back in Chapter 4 I discussed the Billion Dollar Mistake of languages: allowing empty types such as `null` and `undefined` to be assignable to non-empty types. TypeScript's `strictNullChecks` flag roughly adds `| null | undefined` to every type in your code, thereby allowing any variable to receive `null` or `undefined`.

TypeScript best practice is generally to enable `strictNullChecks`. Doing so helps prevent crashes and eliminates the billion dollar mistake.

Refer to Chapter 4 > The Billion Dollar Mistake for more details.

`strictPropertyInitialization`

Back in Chapter 8 I discussed strict initialization checking in classes: making sure that each property on a class is definitely assigned in the class constructor. TypeScript's `strictPropertyInitialization` flag causes a type error to be issued on class properties that have no initializer and are not definitely assigned in the constructor.

TypeScript best practice is generally to enable `strictPropertyInitialization`. Doing so helps prevent crashes from mistakes in class initialization logic.

Refer to Chapter 8 > Initialization Checking for more details.

`useUnknownInCatchVariables`

Error handling in any language is an inherently unsafe concept. Any function can in theory throw any number of errors from edge cases such as reading properties on `undefined` or user-written `` throw`s. In fact, there's no guarantee a thrown error is even an instance of the `Error `` class: code can always `throw "something-else"`.

As a result, TypeScript's default behavior for errors is to give them type `any`: as they could be anything. That allows flexibility in error handling at the cost of relying on the not-very-type-safe `any` by default.

```
try {
  someExternalFunction();
} catch (error) {
  error; // Default type: any
}
```

As with most `any` uses, it would be more technically sound -at the cost of often necessitating explicit type casts or narrowing- to treat errors as `unknown` instead. Catch clause errors are allowed to be annotated as the `any` or `unknown` types.

```
try {
  someExternalFunction();
} catch (error: unknown) {
  error; // Type: unknown
}
```

The strict area flag `useUnknownInCatchVariables` changes TypeScript's default catch clause error type to `unknown`.

```
try {
  someExternalFunction();
} catch (error) {
  error; // Type with useUnknownInCatchVariables: unknown
}
```

Modules

JavaScript's various systems for exporting and importing module contents - AMD, CommonJS, ECMAScript, and so on- are one of the most convoluted module systems in any modern programming language. JavaScript is relatively unusual in that the way files import each others contents is often driven by user-written frameworks such as Webpack. TypeScript does its best to provide configuration options that represent most reasonable user-land module configurations.

Most new TypeScript projects are written with the standardized ECMAScript modules syntax. To recap, here is how ECMAScript modules import a value (`otherValue`) from another module (`"other-module"`) and export their own value (`myValue`):

```
import { otherValue } from "other-module";

export const myValues = ["Hello!", otherValue];
```

`module`

TypeScript provides a `module` compiler option to direct which module system transpiled code will use. When writing source code with

ECMAScript modules, TypeScript may transpile the `export` and `import` statements to a different module system based on the `module` value.

For example, directing that a project written in ECMAScript be output as CommonJS modules in either the command-line:

```
tsc --module commonjs
```

…or in a TSConfig:

```json
{
  "compilerOptions": {
    "module": "commonjs"
  }
}
```

The above code would roughly be output as:

```js
exports.myValues = undefined;
const other_module_1 = require("other-module");
exports.myValues = ["Hello!", other_module_1.otherValue];
```

If your `target` compiler option is `"es3"` or `"es5"`, `module`'s default value will be `"commonjs"`. Otherwise, `module` will default to `"es2015"`.

`moduleResolution`

Module resolution is the process by which the imported path in an import is mapped to a module. TypeScript provides a `moduleResolution` option that you can use to specify the logic for that process. You'll typically want to provide it one of two logic strategies:

- `node`: the behavior used by CommonJS resolvers such as traditional Node.js
- `nodenext`: aligning to the behavior specified for ECMAScript modules

The two strategies are very similar. Most projects could use either of them and not notice a difference.

You can read more on the intricacies behind the scenes of module resolution on *https://www.typescriptlang.org/docs/handbook/module-resolution.html*.

```
tsc --moduleResolution nodenext
```

```
{
  "compilerOptions": {
    "moduleResolution": "nodenext"
  }
}
```

NOTE

For backwards compatibility reasons, TypeScript keeps the default `moduleResolution` value to a `classic` value that was used for projects years ago. You almost certainly do not want the `classic` strategy in any modern project.

JavaScript

TypeScript allows providing `.js` files as inputs to its compiler and type checking. You don't have to write all your source files in TypeScript. Although TypeScript by default ignores files with a `.js` or `.jsx` extension, using its `allowJs` and/or `checkJs` compiler options will allow it to read from, compile, and even -in a limited capacity- type check JavaScript files.

TIP

A common strategy for converting an existing JavaScript projects to TypeScript is to start off with only a few files initially converted to TypeScript. More files may be added over time until there are no more JavaScript files yet. You don't have to go all-in on TypeScript until you're ready to!

`allowJs`

The `allowJs` compiler option allows constructs declared in JavaScript files to factor into type checking TypeScript files When combined with the `jsx` compiler option, `.jsx` files are also allowed.

For example, take this `index.ts` importing a `value` declared in a `values.js` file:

```
// index.ts
import { value } from "./values";

console.log(`I got ${value.toUpperCase()}!`);
```

```
export const value = "temp";
```

Without `allowJs` enabled, the `import` statement would not have a known type. It would be implicitly `any` by default or a type error like `Could not find a declaration file for module "./values".`

TODO: Adjust if https://github.com/microsoft/TypeScript/issues/47189 is accepted

`allowJs` also adds JavaScript files to the list of files compiled to the ECMAScript target and emitted as JavaScript. Source maps and declaration files will be produced as well.

```
tsc --allowJs
```

```
{
  "compilerOptions": {
    "allowJs": true
  }
}
```

`checkJs`

TypeScript can do more than just factor JavaScript files into type checking TypeScript files: it can type check JavaScript files too. The `checkJs` compiler option serves two purposes:

- Defaulting `allowJs` to `true` if it wasn't already
- Enabling the type checker on `.js` and `.jsx` files

Enabling `checkJs` will make TypeScript treat JavaScript files as if they were TypeScript files that don't have any TypeScript-specific syntax. Type mismatches, misspelled variable names, and so on will all cause type errors as they normally would in a TypeScript file.

```
tsc --checkJs
```

```
{
  "compilerOptions": {
    "checkJs": true
  }
}
```

JSDoc Support

Because JavaScript doesn't have TypeScript's rich type syntax, the types of values declared in JavaScript files are often not as precise as those declared in TypeScript files. For example, while TypeScript can infer the value of an object declared as a variable in a JavaScript file, there's no native JavaScript way to type check in that file that the value adheres to any particular `interface`.

I mentioned back in Chapter 1 that the JSDoc community standard provides some ways to describe types using comments. When `allowJs` and/or `checkJs` are enabled, TypeScript will recognize any JSDoc definitions in code.

For example, this snippet declares a function in JSDoc to take in a `text: string[]`. TypeScript can then infer that it returns a `string`. With `checkJs` enabled, TypeScript would know to emit a type error for passing it a `Date` later:

```
// index.js

/**
 * @param {string[]} texts
 */
function upperFirstElement(texts) {
    return texts[0].toUpperCase();
}

upperFirstElement(["temp", "temp"]); // Ok

upperFirstElement(new Date());
//                ~~~~~~~~~~
// Error: Argument of type 'Date' is not
// assignable to parameter of type 'string[]'.
```

The full list of supported JSDoc syntax is available on *https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html*.

Configuration Extensions

As you write more and more TypeScript projects, you may find yourself writing the same project settings repeatedly. Although TypeScript doesn't allow configuration files to be written in JavaScript and use `import` or `require`, it does offer a mechanism for a TSConfig file to opt into "extending", or copying in configuration values, another configuration file.

extends

A TSConfig may extend from another TSConfig with the `extends` configuration option. `extends` takes in a path to another TSConfig file and indicates that all settings from that file should be copied over. It behaves similarly to the `extends` keyword on classes: any option declared on the derived, or child configuration will override any option of the same name on the base, or parent configuration.

For example, many repositories that have multiple TSConfigs, such as monorepos containing multiple `packages/*` directories, by convention create a `tsconfig.base.json` file for `tsconfig.json` files to extend from:

```
// tsconfig.base.json
{
  "compilerOptions": {
    "strict": true
  }
}

// packages/core/tsconfig.json
{
  "extends": "../../tsconfig.base.json",
  "includes": ["src"]
}
```

Note that `compilerOptions` are factored in recursively. Each compiler option from a base TSConfig will copy over to a derived TSConfig unless the derived TSConfig overrides that specific option.

If the above example were to add a TSConfig that adds the `allowJs` option, that new derived TSConfig would still have `compilerOptions.strict` set to `true`:

```
// packages/js/tsconfig.json
{
  "extends": "../../tsconfig.base.json",
  "compilerOptions": {
    "allowJs": true
  },
  "includes": ["src"]
}
```

Extending Modules

The `extends` property may point to either kind of JavaScript import:

- Absolute: starting with `@` or a alphabetical letter

- Relative: a local file path starting with `.`

Absolute `extends` values indicate to extend the TSConfig from an npm module. TypeScript will use the normal Node module resolution system to find a package matching the name. If that package's contains a `tsconfig.json` or its `package.json` contains a `"tsconfig"` field pointing to a different file that exists, that TSConfig will be used.

Many organizations use npm packages to standardize TypeScript compiler options across repositories and/or within monorepos. Shifting the above example TSConfig files to work in a monorepo managed by a tool such as Lerna or Turborepo for a `@my-org` organization:

```
// packages/tsconfig.json
{
  "compilerOptions": {
    "strict": true
  }
}

// packages/core/tsconfig.json
{
  "extends": "@my-org/tsconfig",
  "includes": ["src"]
}

// packages/js/tsconfig.json
{
  "extends": "@my-org/tsconfig",
  "compilerOptions": {
    "allowJs": true
  },
  "includes": ["src"]
}
```

Configuration Bases

Alternately, instead of creating your own configuration from scratch or the `--init` suggestions, you can start with a premade "base" TSConfig file tailored to a particular runtime environment. These premade configuration bases are available on the npm package registry under `@tsconfig/`, such as `@tsconfig/recommended` or `@tsconfig/node16`.

For example, to install the recommended TSConfig base for `create-react-app`:

```
npm install --save-dev @tsconfig/create-react-app
# or
yarn add --dev @tsconfig/create-react-app
```

Once a configuration base package is installed, it can be referenced like any other npm package configuration extension.

```
{
    "extends": "@tsconfig/create-react-app/tsconfig.json"
}
```

The full list of TSConfig bases is documented on *https://github.com/tsconfig/bases*.

TIP

It is generally a good idea to know what TypeScript configuration options your file is using, even if you aren't changing them yourself.

Project References

All the TypeScript configuration files I've shown so far have assumed they manage all the source files of a project. In practice, though, it can be useful in larger projects so use different configuration files for different areas of a project. You may wish to use different compiler options for certain areas of code, or enforce a dependency tree (only allowing certain projects to import files from certain other projects).

TIP

Project references are generally used in larger projects that have multiple distinct areas of code, such as monorepos and modular component systems. You probably don't want to use them for small projects that don't have dozens or more files.

`composite`

To start, TypeScript allows a project define itself as having well-defined inputs and outputs using the `composite` configuration option. When that option is `true`:

- The rootDir setting, if not already explicitly set, defaults to the directory containing the tsconfig file
- All implementation files must be matched by an include pattern or listed in the `files` array.
- `declaration` must be turned on

```
// tsconfig.json
{
  "compilerOptions": {
    "declaration": true
  },
  "composite": true
}
```

These changes help TypeScript enforce that all input files to the project create a matching `.d.ts`. `composite` is generally most useful in combination with the following `references` configuration option.

`references`

A TypeScript project can indicate it relies on the outputs generated by a composite TypeScript project with a `references` setting in its TSConfig. Importing modules from a referenced project will instead load its output `.d.ts` declaration file(s).

```
// core/tsconfig.json
{
  "composite": true
}

// shell/tsconfig.json
{
  "references": [
    { "path": "../core" }
  ]
}
```

NOTE

The `references` configuration option will not be copied from base TSConfigs to derived TSConfigs with via `extends`.

`composite` is generally most useful in combination with the following build mode.

Build Mode

Once an area of code has been set up to use `composite` projects and project `reference`s, it will be possible to use `tsc` in its alternate "build" mode via the `-b`/`--b` CLI flag. Build mode enhances `tsc` into something of a project build coordinator. It lets `tsc` rebuild only the projects that have been changed since the last build, based on when their contents and their file outputs were last generated.

More precisely, TypeScript's build mode will do the following when given a TSConfig:

1. Find that TSConfig's referenced projects
2. Detect if they are up-to-date
3. Build out-of-date projects in the correct order

4. Build the provided TSConfig if it or any of its dependencies have changed

Coordinator Configurations

A common handy pattern for setting up TypeScript project references in a repository is to set up a root-level `tsconfig.json` with an empty `files` array and references to all the project references in the repository. That root TSConfig won't direct TypeScript to build any files itself. Instead it will act purely to tell TypeScript to build referenced projects as needed.

```
// tsconfig.json
{
  "files": [],
  "references": [
    { "path": "./packages/core" },
    { "path": "./packages/shell" }
  ]
}
```

I personally like to standardize on having a script in my `` package.json`s named `build `` or `compile` that calls to `tsc -b` as a shortcut:

```
// package.json
{
  "scripts": {
    "build": "tsc -b"
  }
}
```

Build Mode Options

Build mode supports a few build-specific CLI options:

- `--clean`: Deletes the outputs of the specified projects (may be combined with `--dry`)
- `--dry`: Shows what would be done but doesn't actually build anything

- `--force`: Act as if all projects are out of date
- `-w`/`--watch`: Similar to the typical TypeScript watch mode

Because build mode supports watch mode, running a command like `tsc -b -w` can be a fast way to get an up-to-date listing of all compiler errors in a large project.