O'REILLY®

Effective TypeScript

62 Specific Ways to Improve Your TypeScript

Dan Vanderkam

Praise for *Effective TypeScript*

"*Effective TypeScript* explores the most common questions we see when working with TypeScript and provides practical, results-oriented advice. Regardless of your level of TypeScript experience, you can learn something from this book."

—Ryan Cavanaugh, Engineering Lead for TypeScript at Microsoft

"This book is packed with practical recipes and must be kept on the desk of every professional TypeScript developer. Even if you think you know TypeScript already, get this book and you won't regret it."

—Yakov Fain, Java Champion

"TypeScript is taking over the development world...The deeper understanding of TypeScript this book provides will help many developers shine as they take advantage of TypeScript's powerful features."

—Jason Killian, Cofounder of TypeScript NYC and former TSLint maintainer

"This book is not just about what TypeScript can do—it teaches why each language feature is useful, and where to apply patterns to get the greatest effect. The book focuses on practical advice that will be useful in day-to-day work, with just enough theory to give the reader a deep understanding of how everything works. I consider myself to be an advanced TypeScript user, and I learned a number of new things from this book."

—Jesse Hallett, Senior Software Engineer, Originate, Inc.

Effective TypeScript

62 Specific Ways to Improve Your TypeScript

Dan Vanderkam

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

Effective TypeScript
by Dan Vanderkam

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*.

Acquisitions Editor: Jennifer Pollock
Development Editor: Angela Rufino
Production Editor: Deborah Baker
Copyeditor: Jasmine Kwityn
Proofreader: Kim Wimpsett
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2019: First Edition

Revision History for the First Edition
2019-10-16: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492053743* for release details.

978-1-492-05374-3

[LSI]

For Alex.
You're just my type.

Table of Contents

Preface

In the spring of 2016, I visited my old coworker Evan Martin at Google's San Francisco office and asked him what he was excited about. I'd asked him this same question many times over the years because the answers were wide-ranging and unpredictable but always interesting: C++ build tools, Linux audio drivers, online crosswords, emacs plugins. This time, Evan was excited about TypeScript and Visual Studio Code.

I was surprised! I'd heard of TypeScript before, but I knew only that it was created by Microsoft and that I mistakenly believed it had something to do with .NET. As a lifelong Linux user, I couldn't believe that Evan had hopped on team Microsoft.

Then Evan showed me vscode and the TypeScript playground and I was instantly converted. Everything was so fast, and the code intelligence made it easy to build a mental model of the type system. After years of writing type annotations in JSDoc comments for the Closure Compiler, this felt like typed JavaScript that really worked. And Microsoft had built a cross-platform text editor on top of Chromium? Perhaps this was a language and toolchain worth learning.

I'd recently joined Sidewalk Labs and was writing our first JavaScript. The codebase was still small enough that Evan and I were able to convert it all to TypeScript over the next few days.

I've been hooked ever since. TypeScript is more than just a type system. It also brings a whole suite of language services which are fast and easy to use. The cumulative effect is that TypeScript doesn't just make JavaScript development safer: it also makes it more fun!

Who This Book Is For

The *Effective* books are intended to be the "standard second book" on their topic. You'll get the most out of *Effective TypeScript* if you have some previous practical experience working with JavaScript and TypeScript. My goal with this book is not to teach you TypeScript or JavaScript but to help you advance from a beginning or intermediate user to an expert. The items in this book do this by helping you build mental models of how TypeScript and its ecosystem work, making you aware of pitfalls and traps to avoid, and by guiding you toward using TypeScript's many capabilities in the most effective ways possible. Whereas a reference book will explain the five ways that a language lets you do X, an *Effective* book will tell you which of those five to use and why.

TypeScript has evolved rapidly over the past few years, but my hope is that it has stabilized enough that the content in this book will remain valid for years to come. This book focuses primarily on the language itself, rather than any frameworks or build tools. You won't find any examples of how to use React or Angular with TypeScript, or how to configure TypeScript to work with webpack, Babel, or Rollup. The advice in this book should be relevant to all TypeScript users.

Why I Wrote This Book

When I first started working at Google, I was given a copy of the third edition of *Effective C++*. It was unlike any other programming book I'd read. It made no attempt to be accessible to beginners or to be a complete guide to the language. Rather than telling you what the different features of C++ did, it told you how you should and should not use them. It did so through dozens of short, specific items motivated by concrete examples.

The effect of reading all these examples while using the language daily was unmistakable. I'd used C++ before, but for the first time I felt comfortable with it and knew how to think about the choices it presented me. In later years I would have similar experiences reading *Effective Java* and *Effective JavaScript*.

If you're already comfortable working in a few different programming languages, then diving straight into the odd corners of a new one can be an effective way to challenge your mental models and learn what makes it different. I've learned an enormous amount about TypeScript from writing this book. I hope you'll have the same experience reading it!

How This Book Is Organized

This book is a collection of "items," each of which is a short technical essay that gives you specific advice about some aspect of TypeScript. The items are grouped thematically into chapters, but feel free to jump around and read whichever ones look most interesting to you.

Each item's title conveys the key takeaway. These are the things you should remember as you're using TypeScript, so it's worth skimming the table of contents to get them in your head. If you're writing documentation, for example, and have a nagging sense that you shouldn't be writing type information, then you'll know to go read Item 30: Don't repeat type information in documentation.

The text of the item motivates the advice in the title and backs it up with concrete examples and technical arguments. Almost every point made in this book is demonstrated through example code. I tend to read technical books by looking at the examples and skimming the prose, and I assume you do something similar. I hope you'll read the prose and explanations! But the main points should still come across if you skim the examples.

After reading the item, you should understand why it will help you use TypeScript more effectively. You'll also know enough to understand if it doesn't apply to your situation. Scott Meyers, the author of *Effective C++*, gives a memorable example of this. He met a team of engineers who wrote software that ran on missiles. They knew they could ignore his advice about preventing resource leaks, because their programs would always terminate when the missile hit the target and their hardware blew up. I'm not aware of any missiles with JavaScript runtimes, but the James Webb Space Telescope has one, so you never know!

Finally, each item ends with "Things to Remember." These are a few bullet points that summarize the item. If you're skimming through, you can read these to get a sense for what the item is saying and whether you'd like to read more. You should still read the item! But the summary will do in a pinch.

Conventions in TypeScript Code Samples

All code samples are TypeScript except where it's clear from context that they are JSON, GraphQL, or some other language. Much of the experience of using TypeScript involves interacting with your editor, which presents some challenges in print. I've adopted a few conventions to make this work.

Most editors surface errors using squiggly underlines. To see the full error message, you hover over the underlined text. To indicate an error in a code sample, I put squiggles in a comment line under the place where the error occurs:

```
let str = 'not a number';
let num: number = str;
 // ~~~ Type 'string' is not assignable to type 'number'
```

I occasionally edit the error messages for clarity and brevity, but I never remove an error. If you copy/paste a code sample into your editor, you should get exactly the errors indicated, no more no less.

To draw attention to the lack of an error, I use `// OK`:

```
let str = 'not a number';
let num: number = str as any;  // OK
```

You should be able to hover over a symbol in your editor to see what TypeScript considers its type. To indicate this in text, I use a comment starting with "type is":

```
let v = {str: 'hello', num: 42};  // Type is { str: string; num: number; }
```

The type is for the first symbol on the line (`v` in this case) or for the result of a function call:

```
'four score'.split(' ');  // Type is string[]
```

This matches the type you'd see in your editor character for character. In the case of function calls you may need to assign to a temporary variable to see the type.

I will occasionally introduce no-op statements to indicate the type of a variable on a specific line of code:

```
function foo(x: string|string[]) {
  if (Array.isArray(x)) {
    x;  // Type is string[]
  } else {
    x;  // Type is string
  }
}
```

The `x;` lines are only there to demonstrate the type in each branch of the conditional. You don't need to (and shouldn't) include statements like this in your own code.

Unless it's otherwise noted or clear from context, code samples are intended to be checked with the `--strict` flag. All samples were verified using TypeScript 3.7.0-beta.

Typographical 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.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

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

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

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Effective TypeScript* by Dan Vanderkam (O'Reilly). Copyright 2020 Dan Vanderkam, 978-1-492-05374-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O’Reilly Online Learning

For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live 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, please 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)

You can access the web page for this book, where we list errata, examples, and any additional information, at *https://oreil.ly/Effective_TypeScript*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.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

There are many people who helped make this book possible. Thanks to Evan Martin for introducing me to TypeScript and showing me how to think about it. To Douwe Osinga for connecting me with O’Reilly and being supportive of the project. To Brett Slatkin for advice on structure and for showing me that someone I knew could write

an *Effective* book. To Scott Meyers for coming up with this format and for his "Effective *Effective* Books" blog post, which provided essential guidance.

To my reviewers, Rick Battagline, Ryan Cavanaugh, Boris Cherny, Yakov Fain, Jesse Hallett, and Jason Killian. To all my coworkers at Sidewalk who learned TypeScript with me over the years. To everyone at O'Reilly who helped make this book happen: Angela Rufino, Jennifer Pollock, Deborah Baker, Nick Adams, and Jasmine Kwityn. To the TypeScript NYC crew, Jason, Orta, and Kirill, and to all the speakers. Many items were inspired by talks at the Meetup, as described in the following list:

- Item 3 was inspired by a blog post of Evan Martin's that I found particularly enlightening as I was first learning TypeScript.
- Item 7 was inspired by Anders's talk about structural typing and `keyof` relationships at TSConf 2018, and by a talk of Jesse Hallett's at the April 2019 TypeScript NYC Meetup.
- Both Basarat's guide and helpful answers by DeeV and GPicazo on Stack Overflow were essential in writing Item 9.
- Item 10 builds on similar advice in Item 4 of *Effective JavaScript* (Addison-Wesley).
- I was inspired to write Item 11 by mass confusion around this topic at the August 2019 TypeScript NYC Meetup.
- Item 13 was greatly aided by several questions about `type` vs. `interface` on Stack Overflow. Jesse Hallett suggested the formulation around extensibility.
- Jacob Baskin provided encouragement and early feedback on Item 14.
- Item 19 was inspired by several code samples submitted to the r/typescript subreddit.
- Item 26 is based on my own writing on Medium and a talk I gave at the October 2018 TypeScript NYC Meetup.
- Item 28 is based on common advice in Haskell ("make illegal states unrepresentable"). The Air France 447 story is inspired by Jeff Wise's incredible 2011 article in *Popular Mechanics*.
- Item 29 is based on an issue I ran into with the Mapbox type declarations. Jason Killian suggested the phrasing in the title.
- The advice about naming in Item 36 is common but this particular formulation was inspired by Dan North's short article in *97 Things Every Programmer Should Know* (O'Reilly).
- Item 37 was inspired by Jason Killian's talk at the very first TypeScript NYC Meetup in September 2017.

- Item 41 is based on the TypeScript 2.1 release notes. The term "evolving any" is not widely used outside the TypeScript compiler itself, but I find it useful to have a name for this unusual pattern.
- Item 42 was inspired by a blog post of Jesse Hallett's. Item 43 was greatly aided by feedback from Titian Cernicova Dragomir in TypeScript issue #33128.
- Item 44 is based on York Yao's work on the `type-coverage` tool. I wanted something like this and it existed!
- Item 46 is based on a talk I gave at the December 2017 TypeScript NYC Meetup.
- Item 50 owes a debt of gratitude to David Sheldrick's post on the *Artsy* blog on conditional types, which greatly demystified the topic for me.
- Item 51 was inspired by a talk Steve Faulkner aka southpolesteve gave at the February 2019 Meetup.
- Item 52 is based on my own writing on Medium and work on the typings-checker tool, which eventually got folded into dtslint.
- Item 53 was inspired/reinforced by Kat Busch's Medium post on the various types of enums in TypeScript, as well as Boris Cherny's writings on this topic in *Programming TypeScript* (O'Reilly).
- Item 54 was inspired by my own confusion and that of my coworkers on this topic. The definitive explanation is given by Anders on TypeScript PR #12253.
- The MDN documentation was essential for writing Item 55.
- Item 56 is loosely based on Item 35 of *Effective JavaScript* (Addison-Wesley).
- Chapter 8 is based on my own experience migrating the aging dygraphs library.

I found many of the blog posts and talks that led to this book through the excellent r/typescript subreddit. I'm particularly grateful to developers who provided code samples there which were essential for understanding common issues in beginner TypeScript. Thanks to Marius Schulz for the TypeScript Weekly newsletter. While it's only occasionally weekly, it's always an excellent source of material and a great way to keep up with TypeScript. To Anders, Daniel, Ryan, and the whole TypeScript team at Microsoft for the talks and all the feedback on issues. Most of my issues were misunderstandings, but there is nothing quite so satisfying as filing a bug and immediately seeing Anders Hejlsberg himself fix it! Finally, thanks to Alex for being so supportive during this project and so understanding of all the working vacations, mornings, evenings, and weekends I needed to complete it.

CHAPTER 1
Getting to Know TypeScript

Before we dive into the details, this chapter helps you understand the big picture of TypeScript. What is it and how should you think about it? How does it relate to JavaScript? Are its types nullable or are they not? What's this about `any`? And ducks?

TypeScript is a bit unusual as a language in that it neither runs in an interpreter (as Python and Ruby do) nor compiles down to a lower-level language (as Java and C do). Instead, it compiles to another high-level language, JavaScript. It is this JavaScript that runs, not your TypeScript. So TypeScript's relationship with JavaScript is essential, but it can also be a source of confusion. Understanding this relationship will help you be a more effective TypeScript developer.

TypeScript's type system also has some unusual aspects that you should be aware of. Later chapters cover the type system in much greater detail, but this one will alert you to some of the surprises that it has in store.

Item 1: Understand the Relationship Between TypeScript and JavaScript

If you use TypeScript for long, you'll inevitably hear the phrase "TypeScript is a superset of JavaScript" or "TypeScript is a typed superset of JavaScript." But what does this mean, exactly? And what is the relationship between TypeScript and JavaScript? Since these languages are so closely linked, a strong understanding of how they relate to each is the foundation for using TypeScript well.

TypeScript is a superset of JavaScript in a syntactic sense: so long as your JavaScript program doesn't have any syntax errors then it is also a TypeScript program. It's quite likely that TypeScript's type checker will flag some issues with your code. But this is an independent problem. TypeScript will still parse your code and emit JavaScript. (This is another key part of the relationship. We'll explore this more in Item 3.)

TypeScript files use a *.ts* (or *.tsx*) extension, rather than the *.js* (or *.jsx*) extension of a JavaScript file. This doesn't mean that TypeScript is a completely different language! Since TypeScript is a superset of JavaScript, the code in your *.js* files is already TypeScript. Renaming *main.js* to *main.ts* doesn't change that.

This is enormously helpful if you're migrating an existing JavaScript codebase to TypeScript. It means that you don't have to rewrite any of your code in another language to start using TypeScript and get the benefits it provides. This would not be true if you chose to rewrite your JavaScript in a language like Java. This gentle migration path is one of the best features of TypeScript. There will be much more to say about this topic in Chapter 8.

All JavaScript programs are TypeScript programs, but the converse is not true: there are TypeScript programs which are not JavaScript programs. This is because TypeScript adds additional syntax for specifying types. (There are some other bits of syntax it adds, largely for historical reasons. See Item 53.)

For instance, this is a valid TypeScript program:

```
function greet(who: string) {
  console.log('Hello', who);
}
```

But when you run this through a program like `node` that expects JavaScript, you'll get an error:

```
function greet(who: string) {
                  ^

SyntaxError: Unexpected token :
```

The `: string` is a type annotation that is specific to TypeScript. Once you use one, you've gone beyond plain JavaScript (see Figure 1-1).

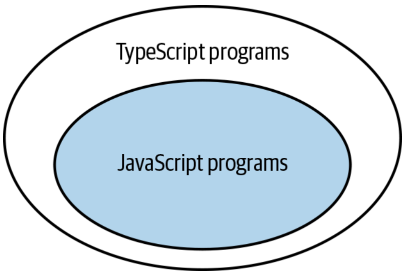


Figure 1-1. All JavaScript is TypeScript, but not all TypeScript is JavaScript

This is not to say that TypeScript doesn't provide value for plain JavaScript programs. It does! For example, this JavaScript program:

```
let city = 'new york city';
console.log(city.toUppercase());
```

will throw an error when you run it:

```
TypeError: city.toUppercase is not a function
```

There are no type annotations in this program, but TypeScript's type checker is still able to spot the problem:

```
let city = 'new york city';
console.log(city.toUppercase());
              // ~~~~~~~~~~~ Property 'toUppercase' does not exist on type
              //             'string'. Did you mean 'toUpperCase'?
```

You didn't have to tell TypeScript that the type of `city` was `string`: it inferred it from the initial value. Type inference is a key part of TypeScript and Chapter 3 explores how to use it well.

One of the goals of TypeScript's type system is to detect code that will throw an exception at runtime, without having to run your code. When you hear TypeScript described as a "static" type system, this is what it refers to. The type checker cannot always spot code that will throw exceptions, but it will try.

Even if your code doesn't throw an exception, it still might not do what you intend. TypeScript tries to catch some of these issues, too. For example, this JavaScript program:

```
const states = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska',  capital: 'Juneau'},
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capitol);
}
```

will log:

```
undefined
undefined
undefined
```

Whoops! What went wrong? This program is valid JavaScript (and hence TypeScript). And it ran without throwing any errors. But it clearly didn't do what you intended. Even without adding type annotations, TypeScript's type checker is able to spot the error (and offer a helpful suggestion):

```
for (const state of states) {
  console.log(state.capitol);
               // ~~~~~~~ Property 'capitol' does not exist on type
               //         '{ name: string; capital: string; }'.
               //         Did you mean 'capital'?
}
```

While TypeScript can catch errors even if you don't provide type annotations, it's able to do a much more thorough job if you do. This is because type annotations tell TypeScript what your *intent* is, and this lets it spot places where your code's behavior does not match your intent. For example, what if you'd reversed the `capital`/`capitol` spelling mistake in the previous example?

```
const states = [
  {name: 'Alabama', capitol: 'Montgomery'},
  {name: 'Alaska',  capitol: 'Juneau'},
  {name: 'Arizona', capitol: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capital);
               // ~~~~~~~ Property 'capital' does not exist on type
               //         '{ name: string; capitol: string; }'.
               //         Did you mean 'capitol'?
}
```

The error that was so helpful before now gets it exactly wrong! The problem is that you've spelled the same property two different ways, and TypeScript doesn't know which one is right. It can guess, but it may not always be correct. The solution is to clarify your intent by explicitly declaring the type of `states`:

```
interface State {
  name: string;
  capital: string;
}
const states: State[] = [
  {name: 'Alabama', capitol: 'Montgomery'},
               // ~~~~~~~~~~~~~~~~~~~~~
  {name: 'Alaska',  capitol: 'Juneau'},
               // ~~~~~~~~~~~~~~~~~
  {name: 'Arizona', capitol: 'Phoenix'},
               // ~~~~~~~~~~~~~~~~~~ Object literal may only specify known
               //         properties, but 'capitol' does not exist in type
               //         'State'.  Did you mean to write 'capital'?
  // ...
];
for (const state of states) {
  console.log(state.capital);
}
```

Now the errors match the problem and the suggested fix is correct. By spelling out our intent, you've also helped TypeScript spot other potential problems. For instance, had you only misspelled `capitol` once in the array, there wouldn't have been an error before. But with the type annotation, there is:

```
const states: State[] = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska',  capitol: 'Juneau'},
                 // ~~~~~~~~~~~~~~~~~ Did you mean to write 'capital'?
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
```

In terms of the Venn diagram, we can add in a new group of programs: TypeScript programs which pass the type checker (see Figure 1-2).

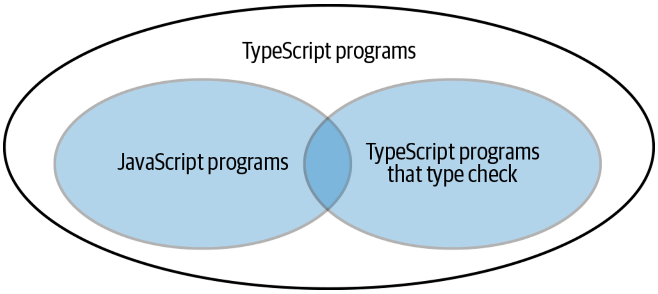


Figure 1-2. All JavaScript programs are TypeScript programs. But only some JavaScript (and TypeScript) programs pass the type checker.

If the statement that "TypeScript is a superset of JavaScript" feels wrong to you, it may be because you're thinking of this third set of programs in the diagram. In practice, this is the most relevant one to the day-to-day experience of using TypeScript. Generally when you use TypeScript, you try to keep your code passing all the type checks.

TypeScript's type system *models* the runtime behavior of JavaScript. This may result in some surprises if you're coming from a language with stricter runtime checks. For example:

```
const x = 2 + '3';  // OK, type is string
const y = '2' + 3;  // OK, type is string
```

These statements both pass the type checker, even though they are questionable and do produce runtime errors in many other languages. But this does model the runtime behavior of JavaScript, where both expressions result in the string `"23"`.

TypeScript does draw the line somewhere, though. The type checker flags issues in all of these statements, even though they do not throw exceptions at runtime:

```
const a = null + 7;  // Evaluates to 7 in JS
       // ~~~~ Operator '+' cannot be applied to types ...
const b = [] + 12;  // Evaluates to '12' in JS
```

```
        // ~~~~~~~ Operator '+' cannot be applied to types ...
alert('Hello', 'TypeScript');  // alerts "Hello"
             // ~~~~~~~~~~~~ Expected 0-1 arguments, but got 2
```

The guiding principle of TypeScript's type system is that it should model JavaScript's runtime behavior. But in all of these cases, TypeScript considers it more likely that the odd usage is the result of an error than the developer's intent, so it goes beyond simply modeling the runtime behavior. We saw another example of this in the `capital`/`capitol` example, where the program didn't throw (it logged `undefined`) but the type checker still flagged an error.

How does TypeScript decide when to model JavaScript's runtime behavior and when to go beyond it? Ultimately this is a matter of taste. By adopting TypeScript you're trusting the judgment of the team that builds it. If you enjoy adding `null` and `7` or `[]` and `12`, or calling functions with superfluous arguments, then TypeScript might not be for you!

If your program type checks, could it still throw an error at runtime? The answer is "yes." Here's an example:

```
const names = ['Alice', 'Bob'];
console.log(names[2].toUpperCase());
```

When you run this, it throws:

```
TypeError: Cannot read property 'toUpperCase' of undefined
```

TypeScript assumed the array access would be within bounds, but it was not. The result was an exception.

Uncaught errors also frequently come up when you use the `any` type, which we'll discuss in Item 5 and in more detail in Chapter 5.

The root cause of these exceptions is that TypeScript's understanding of a value's type and reality have diverged. A type system which can guarantee the accuracy of its static types is said to be *sound*. TypeScript's type system is very much not sound, nor was it ever intended to be. If soundness is important to you, you may want to look at other languages like Reason or Elm. While these do offer more guarantees of runtime safety, this comes at a cost: neither is a superset of JavaScript, so migration will be more complicated.

Things to Remember

- TypeScript is a superset of JavaScript. In other words, all JavaScript programs are already TypeScript programs. TypeScript has some syntax of its own, so TypeScript programs are not, in general, valid JavaScript programs.
- TypeScript adds a type system that models JavaScript's runtime behavior and tries to spot code which will throw exceptions at runtime. But you shouldn't expect it

to flag every exception. It is possible for code to pass the type checker but still throw at runtime.

- While TypeScript's type system largely models JavaScript behavior, there are some constructs that JavaScript allows but TypeScript chooses to bar, such as calling functions with the wrong number of arguments. This is largely a matter of taste.

Item 2: Know Which TypeScript Options You're Using

Does this code pass the type checker?

```
function add(a, b) {
  return a + b;
}
add(10, null);
```

Without knowing which options you're using, it's impossible to say! The TypeScript compiler has an enormous set of these, nearly 100 at the time of this writing.

They can be set via the command line:

```
$ tsc --noImplicitAny program.ts
```

or via a configuration file, *tsconfig.json*:

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

You should prefer the configuration file. It ensures that your coworkers and tools all know exactly how you plan to use TypeScript. You can create one by running `tsc --init`.

Many of TypeScript's configuration settings control where it looks for source files and what sort of output it generates. But a few control core aspects of the language itself. These are high-level design choices that most languages do not leave to their users. TypeScript can feel like a very different language depending on how it is configured. To use it effectively, you should understand the most important of these settings: `noImplicitAny` and `strictNullChecks`.

`noImplicitAny` controls whether variables must have known types. This code is valid when `noImplicitAny` is off:

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

If you mouse over the `add` symbol in your editor, it will reveal what TypeScript has inferred about the type of that function:

```
function add(a: any, b: any): any
```

The `any` types effectively disable the type checker for code involving these parameters. `any` is a useful tool, but it should be used with caution. For much more on `any`, see Item 5 and Chapter 3.

These are called *implicit* `any`*s* because you never wrote the word "any" but still wound up with dangerous `any` types. This becomes an error if you set the `noImplicitAny` option:

```
function add(a, b) {
          // ~    Parameter 'a' implicitly has an 'any' type
          //    ~ Parameter 'b' implicitly has an 'any' type
  return a + b;
}
```

These errors can be fixed by explicitly writing type declarations, either `: any` or a more specific type:

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

TypeScript is the most helpful when it has type information, so you should be sure to set `noImplicitAny` whenever possible. Once you grow accustomed to all variables having types, TypeScript without `noImplicitAny` feels almost like a different language.

For new projects, you should start with `noImplicitAny` on, so that you write the types as you write your code. This will help TypeScript spot problems, improve the readability of your code, and enhance your development experience (see Item 6). Leaving `noImplicitAny` off is only appropriate if you're transitioning a project from JavaScript to TypeScript (see Chapter 8).

`strictNullChecks` controls whether `null` and `undefined` are permissible values in every type.

This code is valid when `strictNullChecks` is off:

```
const x: number = null;  // OK, null is a valid number
```

but triggers an error when you turn `strictNullChecks` on:

```
const x: number = null;
//    ~ Type 'null' is not assignable to type 'number'
```

A similar error would have occurred had you used `undefined` instead of `null`.

If you mean to allow `null`, you can fix the error by making your intent explicit:

```
const x: number | null = null;
```

If you do not wish to permit `null`, you'll need to track down where it came from and add either a check or an assertion:

```
   const el = document.getElementById('status');
   el.textContent = 'Ready';
// ~~ Object is possibly 'null'

   if (el) {
     el.textContent = 'Ready';  // OK, null has been excluded
   }
   el!.textContent = 'Ready';  // OK, we've asserted that el is non-null
```

`strictNullChecks` is tremendously helpful for catching errors involving `null` and `undefined` values, but it does increase the difficulty of using the language. If you're starting a new project, try setting `strictNullChecks`. But if you're new to the language or migrating a JavaScript codebase, you may elect to leave it off. You should certainly set `noImplicitAny` before you set `strictNullChecks`.

If you choose to work without `strictNullChecks`, keep an eye out for the dreaded "undefined is not an object" runtime error. Every one of these is a reminder that you should consider enabling stricter checking. Changing this setting will only get harder as your project grows, so try not to wait too long before enabling it.

There are many other settings that affect language semantics (e.g., `noImplicitThis` and `strictFunctionTypes`), but these are minor compared to `noImplicitAny` and `strictNullChecks`. To enable all of these checks, turn on the `strict` setting. TypeScript is able to catch the most errors with `strict`, so this is where you eventually want to wind up.

Know which options you're using! If a coworker shares a TypeScript example and you're unable to reproduce their errors, make sure your compiler options are the same.

Things to Remember

- The TypeScript compiler includes several settings which affect core aspects of the language.
- Configure TypeScript using *tsconfig.json* rather than command-line options.
- Turn on `noImplicitAny` unless you are transitioning a JavaScript project to TypeScript.
- Use `strictNullChecks` to prevent "undefined is not an object"-style runtime errors.

- Aim to enable `strict` to get the most thorough checking that TypeScript can offer.

Item 3: Understand That Code Generation Is Independent of Types

At a high level, `tsc` (the TypeScript compiler) does two things:

- It converts next-generation TypeScript/JavaScript to an older version of JavaScript that works in browsers ("transpiling").
- It checks your code for type errors.

What's surprising is that these two behaviors are entirely independent of one another. Put another way, the types in your code cannot affect the JavaScript that TypeScript emits. Since it's this JavaScript that gets executed, this means that your types can't affect the way your code runs.

This has some surprising implications and should inform your expectations about what TypeScript can and cannot do for you.

Code with Type Errors Can Produce Output

Because code output is independent of type checking, it follows that code with type errors can produce output!

```
$ cat test.ts
let x = 'hello';
x = 1234;
$ tsc test.ts
test.ts:2:1 - error TS2322: Type '1234' is not assignable to type 'string'

2 x = 1234;
  ~

$ cat test.js
var x = 'hello';
x = 1234;
```

This can be quite surprising if you're coming from a language like C or Java where type checking and output go hand in hand. You can think of all TypeScript errors as being similar to warnings in those languages: it's likely that they indicate a problem and are worth investigating, but they won't stop the build.

Compiling and Type Checking

This is likely the source of some sloppy language that is common around TypeScript. You'll often hear people say that their TypeScript "doesn't compile" as a way of saying that it has errors. But this isn't technically correct! Only the code generation is "compiling." So long as your TypeScript is valid JavaScript (and often even if it isn't), the TypeScript compiler will produce output. It's better to say that your code has errors, or that it "doesn't type check."

Code emission in the presence of errors is helpful in practice. If you're building a web application, you may know that there are problems with a particular part of it. But because TypeScript will still generate code in the presence of errors, you can test the other parts of your application before you fix them.

You should aim for zero errors when you commit code, lest you fall into the trap of having to remember what is an expected or unexpected error. If you want to disable output on errors, you can use the `noEmitOnError` option in *tsconfig.json*, or the equivalent in your build tool.

You Cannot Check TypeScript Types at Runtime

You may be tempted to write code like this:

```
interface Square {
  width: number;
}
interface Rectangle extends Square {
  height: number;
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape instanceof Rectangle) {
                    // ~~~~~~~~~ 'Rectangle' only refers to a type,
                    //           but is being used as a value here
    return shape.width * shape.height;
                    //         ~~~~~~ Property 'height' does not exist
                    //                on type 'Shape'
  } else {
    return shape.width * shape.width;
  }
}
```

The `instanceof` check happens at runtime, but `Rectangle` is a type and so it cannot affect the runtime behavior of the code. TypeScript types are "erasable": part of compilation to JavaScript is simply removing all the `interface`s, `type`s, and type annotations from your code.

To ascertain the type of shape you're dealing with, you'll need some way to reconstruct its type at runtime. In this case you can check for the presence of a `height` property:

```
function calculateArea(shape: Shape) {
  if ('height' in shape) {
    shape;  // Type is Rectangle
    return shape.width * shape.height;
  } else {
    shape;  // Type is Square
    return shape.width * shape.width;
  }
}
```

This works because the property check only involves values available at runtime, but still allows the type checker to refine `shape`'s type to `Rectangle`.

Another way would have been to introduce a "tag" to explicitly store the type in a way that's available at runtime:

```
interface Square {
  kind: 'square';
  width: number;
}
interface Rectangle {
  kind: 'rectangle';
  height: number;
  width: number;
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape.kind === 'rectangle') {
    shape;  // Type is Rectangle
    return shape.width * shape.height;
  } else {
    shape;  // Type is Square
    return shape.width * shape.width;
  }
}
```

The `Shape` type here is an example of a "tagged union." Because they make it so easy to recover type information at runtime, tagged unions are ubiquitous in TypeScript.

Some constructs introduce both a type (which is not available at runtime) and a value (which is). The `class` keyword is one of these. Making `Square` and `Rectangle` classes would have been another way to fix the error:

```
class Square {
  constructor(public width: number) {}
}
class Rectangle extends Square {
  constructor(public width: number, public height: number) {
    super(width);
  }
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape instanceof Rectangle) {
    shape;  // Type is Rectangle
    return shape.width * shape.height;
  } else {
    shape;  // Type is Square
    return shape.width * shape.width;  // OK
  }
}
```

This works because `class Rectangle` introduces both a type and a value, whereas `interface` only introduced a type.

The `Rectangle` in `type Shape = Square | Rectangle` refers to the *type*, but the `Rectangle` in `shape instanceof Rectangle` refers to the *value*. This distinction is important to understand but can be quite subtle. See Item 8.

Type Operations Cannot Affect Runtime Values

Suppose you have a value that could be a string or a number and you'd like to normalize it so that it's always a number. Here's a misguided attempt that the type checker accepts:

```
function asNumber(val: number | string): number {
  return val as number;
}
```

Looking at the generated JavaScript makes it clear what this function really does:

```
function asNumber(val) {
  return val;
}
```

There is no conversion going on whatsoever. The `as number` is a type operation, so it cannot affect the runtime behavior of your code. To normalize the value you'll need to check its runtime type and do the conversion using JavaScript constructs:

```
function asNumber(val: number | string): number {
  return typeof(val) === 'string' ? Number(val) : val;
}
```

(`as number` is a *type assertion*. For more on when it's appropriate to use these, see Item 9.)

Runtime Types May Not Be the Same as Declared Types

Could this function ever hit the final `console.log`?

```
function setLightSwitch(value: boolean) {
  switch (value) {
    case true:
      turnLightOn();
      break;
    case false:
      turnLightOff();
      break;
    default:
      console.log(`I'm afraid I can't do that.`);
  }
}
```

TypeScript usually flags dead code, but it does not complain about this, even with the `strict` option. How could you hit this branch?

The key is to remember that `boolean` is the *declared* type. Because it is a TypeScript type, it goes away at runtime. In JavaScript code, a user might inadvertently call `setLightSwitch` with a value like `"ON"`.

There are ways to trigger this code path in pure TypeScript, too. Perhaps the function is called with a value which comes from a network call:

```
interface LightApiResponse {
  lightSwitchValue: boolean;
}
async function setLight() {
  const response = await fetch('/light');
  const result: LightApiResponse = await response.json();
  setLightSwitch(result.lightSwitchValue);
}
```

You've declared that the result of the `/light` request is `LightApiResponse`, but nothing enforces this. If you misunderstood the API and `lightSwitchValue` is really a `string`, then a string will be passed to `setLightSwitch` at runtime. Or perhaps the API changed after you deployed.

TypeScript can get quite confusing when your runtime types don't match the declared types, and this is a situation you should avoid whenever you can. But be aware that it's possible for a value to have types other than the ones you've declared.

You Cannot Overload a Function Based on TypeScript Types

Languages like C++ allow you to define multiple versions of a function that differ only in the types of their parameters. This is called "function overloading." Because the runtime behavior of your code is independent of its TypeScript types, this construct isn't possible in TypeScript:

```
function add(a: number, b: number) { return a + b; }
      // ~~~ Duplicate function implementation
function add(a: string, b: string) { return a + b; }
      // ~~~ Duplicate function implementation
```

TypeScript *does* provide a facility for overloading functions, but it operates entirely at the type level. You can provide multiple declarations for a function, but only a single implementation:

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;

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

const three = add(1, 2);  // Type is number
const twelve = add('1', '2');  // Type is string
```

The first two declarations of `add` only provide type information. When TypeScript produces JavaScript output, they are removed, and only the implementation remains. (If you use this style of overloading, take a look at Item 50 first. There are some subtleties to be aware of.)

TypeScript Types Have No Effect on Runtime Performance

Because types and type operations are erased when you generate JavaScript, they cannot have an effect on runtime performance. TypeScript's static types are truly zero cost. The next time someone offers runtime overhead as a reason to not use TypeScript, you'll know exactly how well they've tested this claim!

There are two caveats to this:

- While there is no *runtime* overhead, the TypeScript compiler will introduce *build time* overhead. The TypeScript team takes compiler performance seriously and compilation is usually quite fast, especially for incremental builds. If the overhead becomes significant, your build tool may have a "transpile only" option to skip the type checking.
- The code that TypeScript emits to support older runtimes *may* incur a performance overhead vs. native implementations. For example, if you use generator functions and target ES5, which predates generators, then `tsc` will emit some helper code to make things work. This may have some overhead vs. a native implementation of generators. In any case, this has to do with the emit target and language levels and is still independent of the *types*.

Things to Remember

- Code generation is independent of the type system. This means that TypeScript types cannot affect the runtime behavior or performance of your code.
- It is possible for a program with type errors to produce code ("compile").
- TypeScript types are not available at runtime. To query a type at runtime, you need some way to reconstruct it. Tagged unions and property checking are common ways to do this. Some constructs, such as `class`, introduce both a TypeScript type and a value that is available at runtime.

Item 4: Get Comfortable with Structural Typing

JavaScript is inherently duck typed: if you pass a function a value with all the right properties, it won't care how you made the value. It will just use it. ("If it walks like a duck and talks like a duck…") TypeScript models this behavior, and it can sometimes lead to surprising results because the type checker's understanding of a type may be broader than what you had in mind. Having a good grasp of structural typing will help you make sense of errors and non-errors and help you write more robust code.

Say you're working on a physics library and have a 2D vector type:

```
interface Vector2D {
  x: number;
  y: number;
}
```

You write a function to calculate its length:

```
function calculateLength(v: Vector2D) {
  return Math.sqrt(v.x * v.x + v.y * v.y);
}
```

Now you introduce the notion of a named vector:

```
interface NamedVector {
  name: string;
  x: number;
  y: number;
}
```

The `calculateLength` function will work with `NamedVector`s because they have `x` and `y` properties, which are `number`s. TypeScript is smart enough to figure this out:

```
const v: NamedVector = { x: 3, y: 4, name: 'Zee' };
calculateLength(v);  // OK, result is 5
```

What is interesting is that you never declared the relationship between `Vector2D` and `NamedVector`. And you did not have to write an alternative implementation of `calculateLength` calculateLength for `NamedVectors`. TypeScript's type system is modeling JavaScript's runtime behavior (Item 1). It allowed `calculateLength` to be called with a `NamedVector` because its *structure* was compatible with `Vector2D`. This is where the term "structural typing" comes from.

But this can also lead to trouble. Say you add a 3D vector type:

```
interface Vector3D {
  x: number;
  y: number;
  z: number;
}
```

and write a function to normalize them (make their length 1):

```
function normalize(v: Vector3D) {
  const length = calculateLength(v);
  return {
    x: v.x / length,
    y: v.y / length,
    z: v.z / length,
  };
}
```

If you call this function, you're likely to get something longer than unit length:

```
> normalize({x: 3, y: 4, z: 5})
{ x: 0.6, y: 0.8, z: 1 }
```

So what went wrong and why didn't TypeScript catch the error?

The bug is that `calculateLength` operates on 2D vectors but `normalize` operates on 3D vectors. So the `z` component is ignored in the normalization.

What's perhaps more surprising is that the type checker does not catch this issue. Why are you allowed to call `calculateLength` with a 3D vector, despite its type declaration saying that it takes 2D vectors?

What worked so well with named vectors has backfired here. Calling `calculateLength` with an `{x, y, z}` object doesn't throw an error. So the type checker doesn't complain, either, and this behavior has led to a bug. (If you want this to be an error, you have some options. We'll return to this example in Item 37.)

As you write functions, it's easy to imagine that they will be called with arguments having the properties you've declared *and no others*. This is known as a "sealed" or "precise" type, and it cannot be expressed in TypeScript's type system. Like it or not, your types are "open."

This can sometimes lead to surprises:

```
function calculateLengthL1(v: Vector3D) {
  let length = 0;
  for (const axis of Object.keys(v)) {
    const coord = v[axis];
               // ~~~~~~~ Element implicitly has an 'any' type because ...
               //         'string' can't be used to index type 'Vector3D'
    length += Math.abs(coord);
  }
  return length;
}
```

Why is this an error? Since `axis` is one of the keys of `v`, which is a `Vector3D`, it should be either `"x"`, `"y"`, or `"z"`. And according to the declaration of `Vector3D`, these are all `number`s, so shouldn't the type of `coord` be `number`?

Is this error a false positive? No! TypeScript is correct to complain. The logic in the previous paragraph assumes that `Vector3D` is sealed and does not have other properties. But it could:

```
const vec3D = {x: 3, y: 4, z: 1, address: '123 Broadway'};
calculateLengthL1(vec3D);  // OK, returns NaN
```

Since `v` could conceivably have any properties, the type of `axis` is `string`. TypeScript has no reason to believe that `v[axis]` is a number because, as you just saw, it might not be. Iterating over objects can be tricky to type correctly. We'll return to this topic in Item 54, but in this case an implementation without loops would be better:

```
function calculateLengthL1(v: Vector3D) {
  return Math.abs(v.x) + Math.abs(v.y) + Math.abs(v.z);
}
```

Structural typing can also lead to surprises with `class`es, which are compared structurally for assignability:

```
class C {
  foo: string;
  constructor(foo: string) {
    this.foo = foo;
  }
}

const c = new C('instance of C');
const d: C = { foo: 'object literal' };  // OK!
```

Why is `d` assignable to `C`? It has a `foo` property that is a `string`. In addition, it has a `constructor` (from `Object.prototype`) that can be called with one argument (though it is usually called with zero). So the structures match. This might lead to surprises if you have logic in `C`'s constructor and write a function that assumes it's run. This is quite different from languages like C++ or Java, where declaring a parameter of type `C` guarantees that it will be either `C` or a subclass of it.

Structural typing is beneficial when you're writing tests. Say you have a function that runs a query on a database and processes the results:

```
interface Author {
  first: string;
  last: string;
}
function getAuthors(database: PostgresDB): Author[] {
  const authorRows = database.runQuery(`SELECT FIRST, LAST FROM AUTHORS`);
  return authorRows.map(row => ({first: row[0], last: row[1]}));
}
```

To test this, you could create a mock `PostgresDB`. But a better approach is to use structural typing and define a narrower interface:

```
interface DB {
  runQuery: (sql: string) => any[];
}
function getAuthors(database: DB): Author[] {
  const authorRows = database.runQuery(`SELECT FIRST, LAST FROM AUTHORS`);
  return authorRows.map(row => ({first: row[0], last: row[1]}));
}
```

You can still pass `getAuthors` a `PostgresDB` in production since it has a `runQuery` method. Because of structural typing, the `PostgresDB` doesn't need to say that it implements `DB`. TypeScript will figure out that it does.

When you write your tests, you can pass in a simpler object instead:

```
test('getAuthors', () => {
  const authors = getAuthors({
    runQuery(sql: string) {
      return [['Toni', 'Morrison'], ['Maya', 'Angelou']];
    }
  });
  expect(authors).toEqual([
    {first: 'Toni', last: 'Morrison'},
    {first: 'Maya', last: 'Angelou'}
  ]);
});
```

TypeScript will verify that our test `DB` conforms to the interface. And your tests don't need to know anything about your production database: no mocking libraries necessary! By introducing an abstraction (`DB`), we've freed our logic (and tests) from the details of a specific implementation (`PostgresDB`).

Another advantage of structural typing is that it can cleanly sever dependencies between libraries. For more on this, see Item 51.

Things to Remember

- Understand that JavaScript is duck typed and TypeScript uses structural typing to model this: values assignable to your interfaces might have properties beyond those explicitly listed in your type declarations. Types are not "sealed."

- Be aware that classes also follow structural typing rules. You may not have an instance of the class you expect!
- Use structural typing to facilitate unit testing.

Item 5: Limit Use of the any Type

TypeScript's type system is *gradual* and *optional*: *gradual* because you can add types to your code bit by bit and *optional* because you can disable the type checker whenever you like. The key to these features is the `any` type:

```
   let age: number;
   age = '12';
// ~~~ Type '"12"' is not assignable to type 'number'
   age = '12' as any;  // OK
```

The type checker is right to complain here, but you can silence it just by typing `as any`. As you start using TypeScript, it's tempting to use `any` types and type assertions (`as any`) when you don't understand an error, think the type checker is incorrect, or simply don't want to take the time to write out type declarations. In some cases this may be OK, but be aware that `any` eliminates many of the advantages of using TypeScript. You should at least understand its dangers before you use it.

There's No Type Safety with any Types

In the preceding example, the type declaration says that `age` is a `number`. But `any` lets you assign a `string` to it. The type checker will believe that it's a `number` (that's what you said, after all), and the chaos will go uncaught:

```
age += 1;  // OK; at runtime, age is now "121"
```

any Lets You Break Contracts

When you write a function, you are specifying a contract: if the caller gives you a certain type of input, you'll produce a certain type of output. But with an `any` type you can break these contracts:

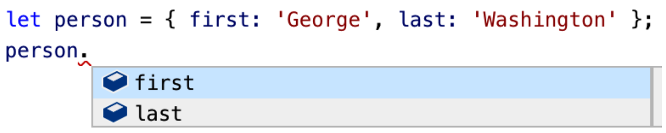
```
function calculateAge(birthDate: Date): number {
  // ...
}

let birthDate: any = '1990-01-19';
calculateAge(birthDate);  // OK
```

The birth date parameter should be a `Date`, not a `string`. The `any` type has let you break the contract of `calculateAge`. This can be particularly problematic because

JavaScript is often willing to implicitly convert between types. A `string` will sometimes work where a `number` is expected, only to break in other circumstances.

There Are No Language Services for any Types

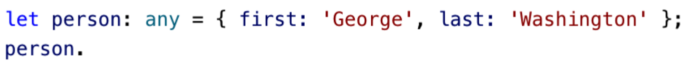
When a symbol has a type, the TypeScript language services are able to provide intelligent autocomplete and contextual documentation (as shown in Figure 1-3).



```
let person = { first: 'George', last: 'Washington' };
person.
```

Figure 1-3. The TypeScript Language Service is able to provide contextual autocomplete for symbols with types.

but for symbols with an `any` type, you're on your own (Figure 1-4).



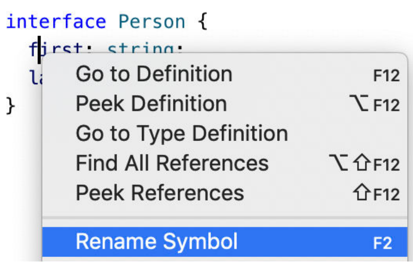
```
let person: any = { first: 'George', last: 'Washington' };
person.
```

Figure 1-4. There is no autocomplete for properties on symbols with any types.

Renaming is another such service. If you have a Person type and functions to format a person's name:

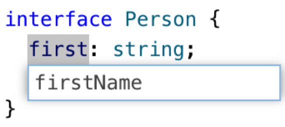
```
interface Person {
  first: string;
  last: string;
}

const formatName = (p: Person) => `${p.first} ${p.last}`;
const formatNameAny = (p: any) => `${p.first} ${p.last}`;
```

then you can select `first` in your editor, choose "Rename Symbol," and change it to `firstName` (see Figures 1-5 and 1-6).



```
interface Person {
  first: string;
  la
}
```

Figure 1-5. Renaming a symbol in vscode.



```
interface Person {
  first: string;
  firstName
}
```

Figure 1-6. Choosing the new name. The TypeScript language service ensures that all uses of the symbol in the project are also renamed.

This changes the `formatName` function but not the `any` version:

```
interface Person {
  firstName: string;
  last: string;
}
const formatName = (p: Person) => `${p.firstName} ${p.last}`;
const formatNameAny = (p: any) => `${p.first} ${p.last}`;
```

TypeScript's motto is "JavaScript that scales." A key part of "scales" is the language services, which are a core part of the TypeScript experience (see Item 6). Losing them will lead to a loss in productivity, not just for you but for everyone else working with your code.

any Types Mask Bugs When You Refactor Code

Suppose you're building a web application in which users can select some sort of item. One of your components might have an `onSelectItem` callback. Writing a type for an Item seems like a hassle, so you just use `any` as a stand-in:

```
interface ComponentProps {
  onSelectItem: (item: any) => void;
}
```

Here's code that manages that component:

```
function renderSelector(props: ComponentProps) { /* ... */ }

let selectedId: number = 0;
```

```
function handleSelectItem(item: any) {
  selectedId = item.id;
}

renderSelector({onSelectItem: handleSelectItem});
```

Later you rework the selector in a way that makes it harder to pass the whole `item` object through to `onSelectItem`. But that's no big deal since you just need the ID. You change the signature in `ComponentProps`:

```
interface ComponentProps {
  onSelectItem: (id: number) => void;
}
```

You update the component and everything passes the type checker. Victory!

…or is it? `handleSelectItem` takes an `any` parameter, so it's just as happy with an Item as it is with an ID. It produces a runtime exception, despite passing the type checker. Had you used a more specific type, this would have been caught by the type checker.

any Hides Your Type Design

The type definition for complex objects like your application state can get quite long. Rather than writing out types for the dozens of properties in your page's state, you may be tempted to just use an `any` type and be done with it.

This is problematic for all the reasons listed in this item. But it's also problematic because it hides the design of your state. As Chapter 4 explains, good type design is essential for writing clean, correct, and understandable code. With an `any` type, your type design is implicit. This makes it hard to know whether the design is a good one, or even what the design is at all. If you ask a coworker to review a change, they'll have to reconstruct whether and how you changed the application state. Better to write it out for everyone to see.

any Undermines Confidence in the Type System

Every time you make a mistake and the type checker catches it, it boosts your confidence in the type system. But when you see a type error at runtime, that confidence takes a hit. If you're introducing TypeScript on a larger team, this might make your coworkers question whether TypeScript is worth the effort. `any` types are often the source of these uncaught errors.

TypeScript aims to make your life easier, but TypeScript with lots of `any` types can be harder to work with than untyped JavaScript because you have to fix type errors *and* still keep track of the real types in your head. When your types match reality, it frees you from the burden of having to keep type information in your head. TypeScript will keep track of it for you.

For the times when you must use `any`, there are better and worse ways to do it. For much more on how to limit the downsides of `any`, see Chapter 5.

Things to Remember

- The `any` type effectively silences the type checker and TypeScript language services. It can mask real problems, harm developer experience, and undermine confidence in the type system. Avoid using it when you can!

CHAPTER 2

TypeScript's Type System

TypeScript generates code (Item 3), but the type system is the main event. This is why you're using the language!

This chapter walks you through the nuts and bolts of TypeScript's type system: how to think about it, how to use it, choices you'll need to make, and features you should avoid. TypeScript's type system is surprisingly powerful and able to express things you might not expect a type system to be able to. The items in this chapter will give you a solid foundation to build upon as you write TypeScript and read the rest of this book.

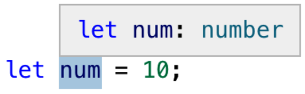
Item 6: Use Your Editor to Interrogate and Explore the Type System

When you install TypeScript, you get two executables:

- `tsc`, the TypeScript compiler
- `tsserver`, the TypeScript standalone server

You're much more likely to run the TypeScript compiler directly, but the server is every bit as important because it provides *language services*. These include autocomplete, inspection, navigation, and refactoring. You typically use these services through your editor. If yours isn't configured to provide them, then you're missing out! Services like autocomplete are one of the things that make TypeScript such a joy to use. But beyond convenience, your editor is the best place to build and test your knowledge of the type system. This will help you build an intuition for when TypeScript is able to infer types, which is key to writing compact, idiomatic code (see Item 19).

The details will vary from editor to editor, but you can generally mouse over a symbol to see what TypeScript considers its type (see Figure 2-1).

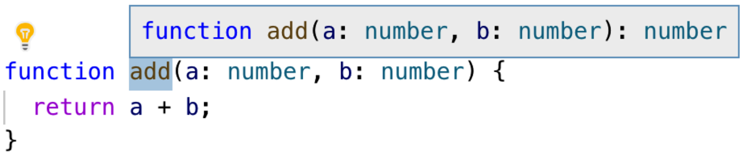


```
let num: number
let num = 10;
```

Figure 2-1. An editor (vscode) showing that the inferred type of the num symbol is number

You didn't write `number` here, but TypeScript was able to figure it out based on the value 10.

You can also inspect functions, as shown in Figure 2-2.

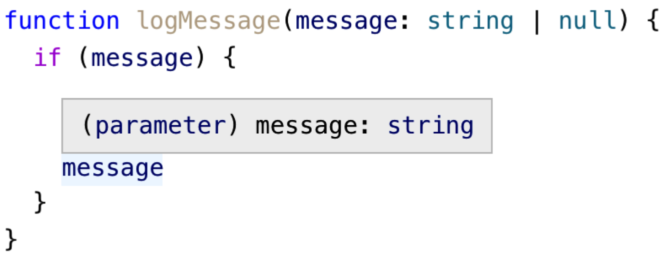


```
function add(a: number, b: number): number
function add(a: number, b: number) {
  return a + b;
}
```

Figure 2-2. Using an editor to reveal the inferred type for a function

The noteworthy bit of information is the inferred value for the return type, `number`. If this does not match your expectation, you should add a type declaration and track down the discrepancy (see Item 9).

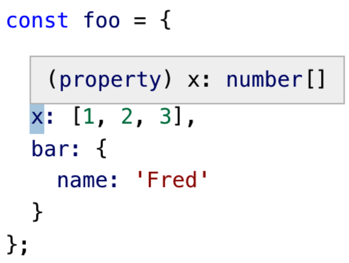
Seeing TypeScript's understanding of a variable's type at any given point is essential for building an intuition around widening (Item 21) and narrowing (Item 22). Seeing the type of a variable change in the branch of a conditional is a tremendous way to build confidence in the type system (see Figure 2-3).



```
function logMessage(message: string | null) {
  if (message) {
    (parameter) message: string
    message
  }
}
```

Figure 2-3. The type of message is string | null outside the branch but string inside.

You can inspect individual properties in a larger object to see what TypeScript has inferred about them (see Figure 2-4).

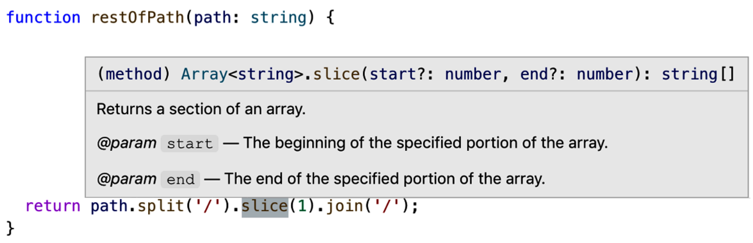


```
const foo = {
  (property) x: number[]
  x: [1, 2, 3],
  bar: {
    name: 'Fred'
  }
};
```

Figure 2-4. Inspecting how TypeScript has inferred types in an object

If your intention was for `x` to be a tuple type (`[number, number, number]`), then a type annotation will be required.

To see inferred generic types in the middle of a chain of operations, inspect the method name (as shown in Figure 2-5).



```
function restOfPath(path: string) {
  (method) Array<string>.slice(start?: number, end?: number): string[]
  Returns a section of an array.
  @param start — The beginning of the specified portion of the array.
  @param end — The end of the specified portion of the array.
  return path.split('/').slice(1).join('/');
}
```

Figure 2-5. Revealing inferred generic types in a chain of method calls

The `Array<string>` indicates that TypeScript understands that `split` produced an array of strings. While there was little ambiguity in this case, this information can prove essential in writing and debugging long chains of function calls.

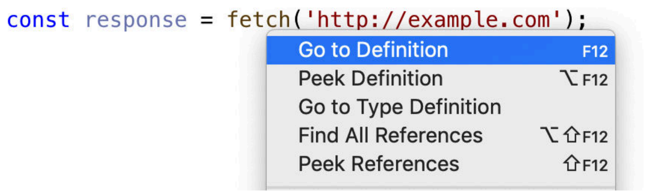
Seeing type errors in your editor can also be a great way to learn the nuances of the type system. For example, this function tries to get an `HTMLElement` by its ID, or return a default one. TypeScript flags two errors:

```
function getElement(elOrId: string|HTMLElement|null): HTMLElement {
  if (typeof elOrId === 'object') {
    return elOrId;
 // ~~~~~~~~~~~~~~ 'HTMLElement | null' is not assignable to 'HTMLElement'
  } else if (elOrId === null) {
    return document.body;
  } else {
    const el = document.getElementById(elOrId);
```

```
      return el;
    // ~~~~~~~~~~ 'HTMLElement | null' is not assignable to 'HTMLElement'
    }
  }
```

The intent in the first branch of the `if` statement was to filter down to just the objects, namely, the `HTMLElement`s. But oddly enough, in JavaScript `typeof null` is `"object"`, so `elOrId` could still be `null` in that branch. You can fix this by putting the `null` check first. The second error is because `document.getElementById` can return `null`, so you need to handle that case as well, perhaps by throwing an exception.

Language services can also help you navigate through libraries and type declarations. Suppose you see a call to the `fetch` function in code and want to learn more about it. Your editor should provide a "Go to Definition" option. In mine it looks like it does in Figure 2-6.



```
const response = fetch('http://example.com');
```

Figure 2-6. The TypeScript language service provides a "Go to Definition" feature that should be surfaced in your editor.

Selecting this option takes you into `lib.dom.d.ts`, the type declarations which TypeScript includes for the DOM:

```
declare function fetch(
  input: RequestInfo, init?: RequestInit
): Promise<Response>;
```

You can see that `fetch` returns a `Promise` and takes two arguments. Clicking through on `RequestInfo` brings you here:

```
type RequestInfo = Request | string;
```

from which you can go to `Request`:

```
declare var Request: {
    prototype: Request;
    new(input: RequestInfo, init?: RequestInit): Request;
};
```

Here you can see that the `Request` type and value are being modeled separately (see Item 8). You've seen `RequestInfo` already. Clicking through on `RequestInit` shows everything you can use to construct a `Request`:

```
interface RequestInit {
    body?: BodyInit | null;
    cache?: RequestCache;
    credentials?: RequestCredentials;
    headers?: HeadersInit;
    // ...
}
```

There are many more types you could follow here, but you get the idea. Type declarations can be challenging to read at first, but they're an excellent way to see what can be done with TypeScript, how the library you're using is modeled, and how you might debug errors. For much more on type declarations, see Chapter 6.

Things to Remember

- Take advantage of the TypeScript language services by using an editor that can use them.
- Use your editor to build an intuition for how the type system works and how TypeScript infers types.
- Know how to jump into type declaration files to see how they model behavior.

Item 7: Think of Types as Sets of Values

At runtime, every variable has a single value chosen from JavaScript's universe of values. There are many possible values, including:

- `42`
- `null`
- `undefined`
- `'Canada'`
- `{animal: 'Whale', weight_lbs: 40_000}`
- `/regex/`
- `new HTMLButtonElement`
- `(x, y) => x + y`

But before your code runs, when TypeScript is checking it for errors, it just has a *type*. This is best thought of as a *set of possible values*. This set is known as the *domain* of the type. For instance, you can think of the `number` type as the set of all number values. `42` and `-37.25` are in it, but `'Canada'` is not. Depending on `strictNullChecks`, `null` and `undefined` may or may not be part of the set.

The smallest set is the empty set, which contains no values. It corresponds to the `never` type in TypeScript. Because its domain is empty, no values are assignable to a variable with a `never` type:

```
const x: never = 12;
   // ~ Type '12' is not assignable to type 'never'
```

The next smallest sets are those which contain single values. These correspond to literal types in TypeScript, also known as unit types:

```
type A = 'A';
type B = 'B';
type Twelve = 12;
```

To form types with two or three values, you can union unit types:

```
type AB = 'A' | 'B';
type AB12 = 'A' | 'B' | 12;
```

and so on. Union types correspond to unions of sets of values.

The word "assignable" appears in many TypeScript errors. In the context of sets of values, it means either "member of" (for a relationship between a value and a type) or "subset of" (for a relationship between two types):

```
const a: AB = 'A';  // OK, value 'A' is a member of the set {'A', 'B'}
const c: AB = 'C';
   // ~ Type '"C"' is not assignable to type 'AB'
```

The type `"C"` is a unit type. Its domain consists of the single value `"C"`. This is not a subset of the domain of `AB` (which consists of the values `"A"` and `"B"`), so this is an error. At the end of the day, almost all the type checker is doing is testing whether one set is a subset of another:

```
// OK, {"A", "B"} is a subset of {"A", "B"}:
const ab: AB = Math.random() < 0.5 ? 'A' : 'B';
const ab12: AB12 = ab;  // OK, {"A", "B"} is a subset of {"A", "B", 12}

declare let twelve: AB12;
const back: AB = twelve;
   // ~~~~ Type 'AB12' is not assignable to type 'AB'
   //        Type '12' is not assignable to type 'AB'
```

The sets for these types are easy to reason about because they are finite. But most types that you work with in practice have infinite domains. Reasoning about these can be harder. You can think of them as either being built constructively:

```
type Int = 1 | 2 | 3 | 4 | 5 // | ...
```

or by describing their members:

```
interface Identified {
  id: string;
}
```

Think of this interface as a description of the values in the domain of its type. Does the value have an `id` property whose value is assignable to (a member of) `string`? Then it's an `Identifiable`.

That's *all* it says. As Item 4 explained, TypeScript's structural typing rules mean that the value could have other properties, too. It could even be callable! This fact can sometimes be obscured by excess property checking (see Item 11).

Thinking of types as sets of values helps you reason about operations on them. For example:

```
interface Person {
  name: string;
}
interface Lifespan {
  birth: Date;
  death?: Date;
}
type PersonSpan = Person & Lifespan;
```

The `&` operator computes the intersection of two types. What sorts of values belong to the `PersonSpan` type? On first glance the `Person` and `Lifespan` interfaces have no properties in common, so you might expect it to be the empty set (i.e., the `never` type). But type operations apply to the sets of values (the domain of the type), not to the properties in the interface. And remember that values with additional properties still belong to a type. So a value that has the properties of *both* `Person` *and* `Lifespan` will belong to the intersection type:

```
const ps: PersonSpan = {
  name: 'Alan Turing',
  birth: new Date('1912/06/23'),
  death: new Date('1954/06/07'),
};  // OK
```

Of course, a value could have more than those three properties and still belong to the type! The general rule is that values in an intersection type contain the union of properties in each of its constituents.

The intuition about intersecting properties is correct, but for the *union* of two interfaces, rather than their intersection:

```
type K = keyof (Person | Lifespan);  // Type is never
```

There are no keys that TypeScript can guarantee belong to a value in the union type, so `keyof` for the union must be the empty set (`never`). Or, more formally:

```
keyof (A&B) = (keyof A) | (keyof B)
keyof (A|B) = (keyof A) & (keyof B)
```

If you can build an intuition for why these equations hold, you'll have come a long way toward understanding TypeScript's type system!

Another perhaps more common way to write the `PersonSpan` type would be with `extends`:

```
interface Person {
  name: string;
}
interface PersonSpan extends Person {
  birth: Date;
  death?: Date;
}
```

Thinking of types as sets of values, what does `extends` mean? Just like "assignable to," you can read it as "subset of." Every value in `PersonSpan` must have a `name` property which is a `string`. And every value must also have a `birth` property, so it's a proper subset.

You might hear the term "subtype." This is another way of saying that one set's domain is a subset of the others. Thinking in terms of one-, two-, and three-dimensional vectors:

```
interface Vector1D { x: number; }
interface Vector2D extends Vector1D { y: number; }
interface Vector3D extends Vector2D { z: number; }
```

You'd say that a `Vector3D` is a subtype of `Vector2D`, which is a subtype of `Vector1D` (in the context of classes you'd say "subclass"). This relationship is usually drawn as a hierarchy, but thinking in terms of sets of values, a Venn diagram is more appropriate (see Figure 2-7).

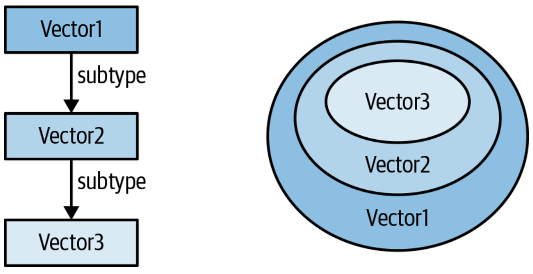


Figure 2-7. Two ways of thinking of type relationships: as a hierarchy or as overlapping sets

With the Venn diagram, it's clear that the subset/subtype/assignability relationships are unchanged if you rewrite the interfaces without `extends`:

```
interface Vector1D { x: number; }
interface Vector2D { x: number; y: number; }
interface Vector3D { x: number; y: number; z: number; }
```

The sets haven't changed, so neither has the Venn diagram.

While both interpretations are workable for object types, the set interpretation becomes much more intuitive when you start thinking about literal types and union types. `extends` can also appear as a constraint in a generic type, and it also means "subset of" in this context (Item 14):

```
function getKey<K extends string>(val: any, key: K) {
  // ...
}
```

What does it mean to extend `string`? If you're used to thinking in terms of object inheritance, it's hard to interpret. You could define a subclass of the object wrapper type `String` (Item 10), but that seems inadvisable.

Thinking in terms of sets, on the other hand, it's crystal clear: any type whose domain is a subset of `string` will do. This includes string literal types, unions of string literal types and `string` itself:

```
getKey({}, 'x');  // OK, 'x' extends string
getKey({}, Math.random() < 0.5 ? 'a' : 'b');  // OK, 'a'|'b' extends string
getKey({}, document.title);  // OK, string extends string
getKey({}, 12);
        // ~~ Type '12' is not assignable to parameter of type 'string'
```

"extends" has turned into "assignable" in the last error, but this shouldn't trip us up since we know to read both as "subset of." This is also a helpful mindset with finite sets, such the ones you might get from `keyof T`, which returns type for just the keys of an object type:

```
interface Point {
  x: number;
  y: number;
}
type PointKeys = keyof Point;  // Type is "x" | "y"

function sortBy<K extends keyof T, T>(vals: T[], key: K): T[] {
  // ...
}
const pts: Point[] = [{x: 1, y: 1}, {x: 2, y: 0}];
sortBy(pts, 'x');  // OK, 'x' extends 'x'|'y' (aka keyof T)
sortBy(pts, 'y');  // OK, 'y' extends 'x'|'y'
sortBy(pts, Math.random() < 0.5 ? 'x' : 'y');  // OK, 'x'|'y' extends 'x'|'y'
sortBy(pts, 'z');
         // ~~~ Type '"z"' is not assignable to parameter of type '"x" | "y"
```

The set interpretation also makes more sense when you have types whose relationship isn't strictly hierarchical. What's the relationship between `string|number` and `string|Date`, for instance? Their intersection is non-empty (it's `string`), but neither is a subset of the other. The relationship between their domains is clear, even though these types don't fit into a strict hierarchy (see Figure 2-8).

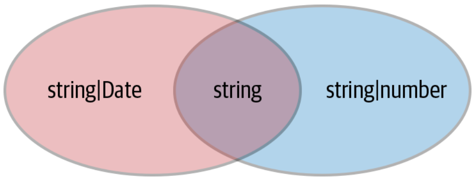


Figure 2-8. Union types may not fit into a hierarchy but can be thought of in terms of sets of values.

Thinking of types as sets can also clarify the relationships between arrays and tuples. For example:

```
const list = [1, 2];  // Type is number[]
const tuple: [number, number] = list;
   // ~~~~~ Type 'number[]' is missing the following
   //       properties from type '[number, number]': 0, 1
```

Are there lists of numbers which are not pairs of numbers? Sure! The empty list and the list `[1]` are examples. It therefore makes sense that `number[]` is not assignable to `[number, number]` since it's not a subset of it. (The reverse assignment does work.)

Is a triple assignable to a pair? Thinking in terms of structural typing, you might expect it to be. A pair has `0` and `1` keys, so mightn't it have others, too, like `2`?

```
const triple: [number, number, number] = [1, 2, 3];
const double: [number, number] = triple;
   // ~~~~~~ '[number, number, number]' is not assignable to '[number, number]'
   //          Types of property 'length' are incompatible
   //          Type '3' is not assignable to type '2'
```

The answer is "no," and for an interesting reason. Rather than modeling a pair of numbers as `{0: number, 1: number}`, TypeScript models it as `{0: number, 1: number, length: 2}`. This makes sense—you can check the length of a tuple—and it precludes this assignment. And that's probably for the best!

If types are best thought of as sets of values, that means that two types with the same sets of values are the same. And indeed this is true. Unless two types are semantically different and just happen to have the same domain, there's no reason to define the same type twice.

Finally, it's worth noting that not all sets of values correspond to TypeScript types. There is no TypeScript type for all the integers, or for all the objects that have `x` and `y` properties but no others. You can sometimes subtract types using `Exclude`, but only when it would result in a proper TypeScript type:

```
type T = Exclude<string|Date, string|number>;  // Type is Date
type NonZeroNums = Exclude<number, 0>;  // Type is still just number
```

Table 2-1 summarizes the correspondence between TypeScript terms and terms from set theory.

Table 2-1. TypeScript terms and set terms

TypeScript term	Set term
`never`	∅ (empty set)
Literal type	Single element set
Value assignable to T	Value ∈ T (member of)
T1 assignable to T2	T1 ⊆ T2 (subset of)
T1 extends T2	T1 ⊆ T2 (subset of)
T1 | T2	T1 ∪ T2 (union)
T1 & T2	T1 ∩ T2 (intersection)
`unknown`	Universal set

Things to Remember

- Think of types as sets of values (the type's *domain*). These sets can either be finite (e.g., `boolean` or literal types) or infinite (e.g., `number` or `string`).
- TypeScript types form intersecting sets (a Venn diagram) rather than a strict hierarchy. Two types can overlap without either being a subtype of the other.
- Remember that an object can still belong to a type even if it has additional properties that were not mentioned in the type declaration.
- Type operations apply to a set's domain. The intersection of `A` and `B` is the intersection of `A`'s domain and `B`'s domain. For object types, this means that values in `A & B` have the properties of both `A` and `B`.
- Think of "extends," "assignable to," and "subtype of" as synonyms for "subset of."

Item 8: Know How to Tell Whether a Symbol Is in the Type Space or Value Space

A symbol in TypeScript exists in one of two spaces:

- Type space
- Value space

This can get confusing because the same name can refer to different things depending on which space it's in:

```
interface Cylinder {
  radius: number;
  height: number;
}

const Cylinder = (radius: number, height: number) => ({radius, height});
```

`interface Cylinder` introduces a symbol in type space. `const Cylinder` introduces a symbol with the same name in value space. They have nothing to do with one another. Depending on the context, when you type `Cylinder`, you'll either be referring to the type or the value. Sometimes this can lead to errors:

```
function calculateVolume(shape: unknown) {
  if (shape instanceof Cylinder) {
    shape.radius
       // ~~~~~~ Property 'radius' does not exist on type '{}'
  }
}
```

What's going on here? You probably intended the `instanceof` to check whether the shape was of the `Cylinder` type. But `instanceof` is JavaScript's runtime operator, and it operates on values. So `instanceof Cylinder` refers to the function, not the type.

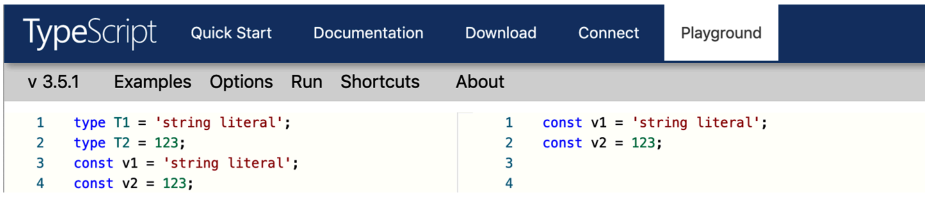
It's not always obvious at first glance whether a symbol is in type space or value space. You have to tell from the context in which the symbol occurs. This can get especially confusing because many type-space constructs look exactly the same as value-space constructs.

Literals, for example:

```
type T1 = 'string literal';
type T2 = 123;
const v1 = 'string literal';
const v2 = 123;
```

Generally the symbols after a `type` or `interface` are in type space while those introduced in a `const` or `let` declaration are values.

One of the best ways to build an intuition for the two spaces is through the TypeScript Playground, which shows you the generated JavaScript for your TypeScript source. Types are erased during compilation (Item 3), so if a symbol disappears then it was probably in type space (see Figure 2-9).



```
type T1 = 'string literal';
type T2 = 123;
const v1 = 'string literal';
const v2 = 123;
```

```
const v1 = 'string literal';
const v2 = 123;
```

Figure 2-9. The TypeScript playground showing generated JavaScript. The symbols on the first two lines go away, so they were in type space.

Statements in TypeScript can alternate between type space and value space. The symbols after a type declaration (`:`) or an assertion (`as`) are in type space while everything after an = is in value space. For example:

```
interface Person {
  first: string;
  last: string;
}
const p: Person = { first: 'Jane', last: 'Jacobs' };
//    -           ---------------------------------- Values
//       ------ Type
```

Function statements in particular can alternate repeatedly between the spaces:

```
function email(p: Person, subject: string, body: string): Response {
  //     ----- -          -------          ----  Values
  //              ------           ------       ------   -------- Types
  // ...
}
```

The `class` and `enum` constructs introduce both a type and a value. In the first example, `Cylinder` should have been a `class`:

```
class Cylinder {
  radius=1;
  height=1;
}

function calculateVolume(shape: unknown) {
  if (shape instanceof Cylinder) {
    shape  // OK, type is Cylinder
    shape.radius  // OK, type is number
  }
}
```

The TypeScript type introduced by a class is based on its shape (its properties and methods) while the value is the constructor.

There are many operators and keywords that mean different things in a type or value context. `typeof`, for instance:

```
type T1 = typeof p;  // Type is Person
type T2 = typeof email;
    // Type is (p: Person, subject: string, body: string) => Response

const v1 = typeof p;  // Value is "object"
const v2 = typeof email;  // Value is "function"
```

In a type context, `typeof` takes a value and returns its TypeScript type. You can use these as part of a larger type expression, or use a `type` statement to give them a name.

In a value context, `typeof` is JavaScript's runtime `typeof` operator. It returns a string containing the runtime type of the symbol. This is *not* the same as the TypeScript type! JavaScript's runtime type system is much simpler than TypeScript's static type system. In contrast to the infinite variety of TypeScript types, there have historically only been six runtime types in JavaScript: "string," "number," "boolean," "undefined," "object," and "function."

`typeof` always operates on values. You can't apply it to types. The `class` keyword introduces both a value and a type, so what is the `typeof` a class? It depends on the context:

```
const v = typeof Cylinder;  // Value is "function"
type T = typeof Cylinder;  // Type is typeof Cylinder
```

The value is `"function"` because of how classes are implemented in JavaScript. The type isn't particularly illuminating. What's important is that it's *not* `Cylinder` (the type of an instance). It's actually the constructor function, which you can see by using it with `new`:

```
declare let fn: T;
const c = new fn();  // Type is Cylinder
```

You can go between the constructor type and the instance type using the `InstanceType` generic:

```
type C = InstanceType<typeof Cylinder>;  // Type is Cylinder
```

The `[]` property accessor also has an identical-looking equivalent in type space. But be aware that while `obj['field']` and `obj.field` are equivalent in value space, they are not in type space. You must use the former to get the type of another type's property:

```
const first: Person['first'] = p['first'];  // Or p.first
   // -----                    ---------- Values
   //        ------ -------  Types
```

`Person['first']` is a *type* here since it appears in a type context (after a `:`). You can put any type in the index slot, including union types or primitive types:

```
type PersonEl = Person['first' | 'last'];  // Type is string
type Tuple = [string, number, Date];
type TupleEl = Tuple[number];  // Type is string | number | Date
```

See Item 14 for more on this.

There are many other constructs that have different meanings in the two spaces:

- `this` in value space is JavaScript's `this` keyword (Item 49). As a type, `this` is the TypeScript type of `this`, aka "polymorphic this." It's helpful for implementing method chains with subclasses.
- In value space `&` and `|` are bitwise AND and OR. In type space they are the intersection and union operators.
- `const` introduces a new variable, but `as const` changes the inferred type of a literal or literal expression (Item 21).
- `extends` can define a subclass (`class A extends B`) or a subtype (`interface A extends B`) or a constraint on a generic type (`Generic<T extends number>`).
- `in` can either be part of a loop (`for (key in object)`) or a mapped type (Item 14).

If TypeScript doesn't seem to understand your code at all, it may be because of confusion around type and value space. For example, say you change the `email` function from earlier to take its arguments in a single object parameter:

```
function email(options: {person: Person, subject: string, body: string}) {
  // ...
}
```

In JavaScript you can use destructuring assignment to create local variables for each property in the object:

```
function email({person, subject, body}) {
  // ...
}
```

If you try to do the same in TypeScript, you get some confusing errors:

```
function email({
  person: Person,
       // ~~~~~~ Binding element 'Person' implicitly has an 'any' type
  subject: string,
        // ~~~~~~ Duplicate identifier 'string'
        //        Binding element 'string' implicitly has an 'any' type
  body: string}
     // ~~~~~~ Duplicate identifier 'string'
     //        Binding element 'string' implicitly has an 'any' type
) { /* ... */ }
```

The problem is that `Person` and `string` are being interpreted in a value context. You're trying to create a variable named `Person` and two variables named `string`. Instead, you should separate the types and values:

```
function email(
  {person, subject, body}: {person: Person, subject: string, body: string}
) {
  // ...
}
```

This is significantly more verbose, but in practice you may have a named type for the parameters or be able to infer them from context (Item 26).

While the similar constructs in type and value can be confusing at first, they're eventually useful as a mnemonic once you get the hang of it.

Things to Remember

- Know how to tell whether you're in type space or value space while reading a TypeScript expression. Use the TypeScript playground to build an intuition for this.
- Every value has a type, but types do not have values. Constructs such as `type` and `interface` exist only in the type space.
- `"foo"` might be a string literal, or it might be a string literal type. Be aware of this distinction and understand how to tell which it is.
- `typeof`, `this`, and many other operators and keywords have different meanings in type space and value space.
- Some constructs such as `class` or `enum` introduce both a type and a value.

Item 9: Prefer Type Declarations to Type Assertions

TypeScript seems to have two ways of assigning a value to a variable and giving it a type:

```
interface Person { name: string };

const alice: Person = { name: 'Alice' };  // Type is Person
const bob = { name: 'Bob' } as Person;  // Type is Person
```

While these achieve similar ends, they are actually quite different! The first (`alice: Person`) adds a *type declaration* to the variable and ensures that the value conforms to the type. The latter (`as Person`) performs a *type assertion*. This tells TypeScript that, despite the type it inferred, you know better and would like the type to be `Person`.

In general, you should prefer type declarations to type assertions. Here's why:

```
const alice: Person = {};
   // ~~~~~ Property 'name' is missing in type '{}'
   //       but required in type 'Person'
const bob = {} as Person;  // No error
```

The type declaration verifies that the value conforms to the interface. Since it does not, TypeScript flags an error. The type assertion silences this error by telling the type checker that, for whatever reason, you know better than it does.

The same thing happens if you specify an additional property:

```
const alice: Person = {
  name: 'Alice',
  occupation: 'TypeScript developer'
// ~~~~~~~~~ Object literal may only specify known properties
//           and 'occupation' does not exist in type 'Person'
};
const bob = {
  name: 'Bob',
  occupation: 'JavaScript developer'
} as Person;  // No error
```

This is excess property checking at work (Item 11), but it doesn't apply if you use an assertion.

Because they provide additional safety checks, you should use type declarations unless you have a specific reason to use a type assertion.

You may also see code that looks like `const bob = <Person>{}`. This was the original syntax for assertions and is equivalent to `{} as Person`. It is less common now because `<Person>` is interpreted as a start tag in *.tsx* files (TypeScript + React).

It's not always clear how to use a declaration with arrow functions. For example, what if you wanted to use the named `Person` interface in this code?

```
const people = ['alice', 'bob', 'jan'].map(name => ({name}));
// { name: string; }[]... but we want Person[]
```

It's tempting to use a type assertion here, and it seems to solve the problem:

```
const people = ['alice', 'bob', 'jan'].map(
  name => ({name} as Person)
); // Type is Person[]
```

But this suffers from all the same issues as a more direct use of type assertions. For example:

```
const people = ['alice', 'bob', 'jan'].map(name => ({} as Person));
// No error
```

So how do you use a type declaration in this context instead? The most straightforward way is to declare a variable in the arrow function:

```
const people = ['alice', 'bob', 'jan'].map(name => {
  const person: Person = {name};
  return person
}); // Type is Person[]
```

But this introduces considerable noise compared to the original code. A more concise way is to declare the return type of the arrow function:

```
const people = ['alice', 'bob', 'jan'].map(
  (name): Person => ({name})
); // Type is Person[]
```

This performs all the same checks on the value as the previous version. The parentheses are significant here! `(name): Person` infers the type of `name` and specifies that the returned type should be `Person`. But `(name: Person)` would specify the type of `name` as `Person` and allow the return type to be inferred, which would produce an error.

In this case you could have also written the final desired type and let TypeScript check the validity of the assignment:

```
const people: Person[] = ['alice', 'bob', 'jan'].map(
  (name): Person => ({name})
);
```

But in the context of a longer chain of function calls it may be necessary or desirable to have the named type in place earlier. And it will help flag errors where they occur.

So when *should* you use a type assertion? Type assertions make the most sense when you truly do know more about a type than TypeScript does, typically from context that isn't available to the type checker. For instance, you may know the type of a DOM element more precisely than TypeScript does:

```
document.querySelector('#myButton').addEventListener('click', e => {
  e.currentTarget // Type is EventTarget
  const button = e.currentTarget as HTMLButtonElement;
  button // Type is HTMLButtonElement
});
```

Because TypeScript doesn't have access to the DOM of your page, it has no way of knowing that `#myButton` is a button element. And it doesn't know that the `currentTarget` of the event should be that same button. Since you have information that TypeScript does not, a type assertion makes sense here. For more on DOM types, see Item 55.

You may also run into the non-null assertion, which is so common that it gets a special syntax:

```
const elNull = document.getElementById('foo');  // Type is HTMLElement | null
const el = document.getElementById('foo')!; // Type is HTMLElement
```

Used as a prefix, `!` is boolean negation. But as a suffix, `!` is interpreted as an assertion that the value is non-null. You should treat `!` just like any other assertion: it is erased during compilation, so you should only use it if you have information that the type checker lacks and can ensure that the value is non-null. If you can't, you should use a conditional to check for the `null` case.

Type assertions have their limits: they don't let you convert between arbitrary types. The general idea is that you can use a type assertion to convert between A and B if either is a subset of the other. `HTMLElement` is a subtype of `HTMLElement | null`, so this type assertion is OK. `HTMLButtonElement` is a subtype of `EventTarget`, so that was OK, too. And `Person` is a subtype of `{}`, so that assertion is also fine.

But you can't convert between a `Person` and an `HTMLElement` since neither is a subtype of the other:

```
interface Person { name: string; }
const body = document.body;
const el = body as Person;
        // ~~~~~~~~~~~~~~ Conversion of type 'HTMLElement' to type 'Person'
        //                may be a mistake because neither type sufficiently
        //                overlaps with the other. If this was intentional,
        //                convert the expression to 'unknown' first
```

The error suggests an escape hatch, namely, using the `unknown` type (Item 42). Every type is a subtype of `unknown`, so assertions involving `unknown` are always OK. This lets you convert between arbitrary types, but at least you're being explicit that you're doing something suspicious!

```
const el = document.body as unknown as Person;  // OK
```

Things to Remember

- Prefer type declarations (`: Type`) to type assertions (`as Type`).
- Know how to annotate the return type of an arrow function.
- Use type assertions and non-null assertions when you know something about types that TypeScript does not.

Item 10: Avoid Object Wrapper Types (String, Number, Boolean, Symbol, BigInt)

In addition to objects, JavaScript has seven types of primitive values: strings, numbers, booleans, `null`, `undefined`, symbol, and bigint. The first five have been around since the beginning. The symbol primitive was added in ES2015, and bigint is in the process of being finalized.

Primitives are distinguished from objects by being immutable and not having methods. You might object that strings *do* have methods:

```
> 'primitive'.charAt(3)
"m"
```

But things are not quite as they seem. There's actually something surprising and subtle going on here. While a string *primitive* does not have methods, JavaScript also defines a `String` *object* type that does. JavaScript freely converts between these types. When you access a method like `charAt` on a string primitive, JavaScript wraps it in a `String` object, calls the method, and then throws the object away.

You can observe this if you monkey-patch `String.prototype` (Item 43):

```
// Don't do this!
const originalCharAt = String.prototype.charAt;
String.prototype.charAt = function(pos) {
  console.log(this, typeof this, pos);
  return originalCharAt.call(this, pos);
};
console.log('primitive'.charAt(3));
```

This produces the following output:

```
[String: 'primitive'] 'object' 3
m
```

The `this` value in the method is a `String` object wrapper, not a string primitive. You can instantiate a `String` object directly and it will sometimes behave like a string primitive. But not always. For example, a `String` object is only ever equal to itself:

```
> "hello" === new String("hello")
false
> new String("hello") === new String("hello")
false
```

The implicit conversion to object wrapper types explains an odd phenomenon in JavaScript—if you assign a property to a primitive, it disappears:

```
> x = "hello"
> x.language = 'English'
'English'
> x.language
undefined
```

Now you know the explanation: `x` is converted to a `String` instance, the `language` property is set on that, and then the object (with its `language` property) is thrown away.

There are object wrapper types for the other primitives as well: `Number` for numbers, `Boolean` for booleans, `Symbol` for symbols, and `BigInt` for bigints (there are no object wrappers for `null` and `undefined`).

These wrapper types exist as a convenience to provide methods on the primitive values and to provide static methods (e.g., `String.fromCharCode`). But there's usually no reason to instantiate them directly.

TypeScript models this distinction by having distinct types for the primitives and their object wrappers:

- `string` and `String`
- `number` and `Number`
- `boolean` and `Boolean`
- `symbol` and `Symbol`
- `bigint` and `BigInt`

It's easy to inadvertently type `String` (especially if you're coming from Java or C#) and it even seems to work, at least initially:

```
function getStringLen(foo: String) {
  return foo.length;
}

getStringLen("hello");  // OK
getStringLen(new String("hello"));  // OK
```

But things go awry when you try to pass a `String` object to a method that expects a `string`:

```
function isGreeting(phrase: String) {
  return [
    'hello',
    'good day'
  ].includes(phrase);
          // ~~~~~~
          // Argument of type 'String' is not assignable to parameter
          // of type 'string'.
          // 'string' is a primitive, but 'String' is a wrapper object;
          // prefer using 'string' when possible
}
```

So `string` is assignable to `String`, but `String` is not assignable to `string`. Confusing? Follow the advice in the error message and stick with `string`. All the type declarations that ship with TypeScript use it, as do the typings for almost all other libraries.

Another way you can wind up with wrapper objects is if you provide an explicit type annotation with a capital letter:

```
const s: String = "primitive";
const n: Number = 12;
const b: Boolean = true;
```

Of course, the values at runtime are still primitives, not objects. But TypeScript permits these declarations because the primitive types are assignable to the object wrappers. These annotations are both misleading and redundant (Item 19). Better to stick with the primitive types.

As a final note, it's OK to call `BigInt` and `Symbol` without `new`, since these create primitives:

```
> typeof BigInt(1234)
"bigint"
> typeof Symbol('sym')
"symbol"
```

These are the `BigInt` and `Symbol` *values*, not the TypeScript types (Item 8). Calling them results in values of type `bigint` and `symbol`.

Things to Remember

- Understand how object wrapper types are used to provide methods on primitive values. Avoid instantiating them or using them directly.
- Avoid TypeScript object wrapper types. Use the primitive types instead: `string` instead of `String`, `number` instead of `Number`, `boolean` instead of `Boolean`, `symbol` instead of `Symbol`, and `bigint` instead of `BigInt`.

Item 11: Recognize the Limits of Excess Property Checking

When you assign an object literal to a variable with a declared type, TypeScript makes sure it has the properties of that type *and no others*:

```
interface Room {
  numDoors: number;
  ceilingHeightFt: number;
}
const r: Room = {
  numDoors: 1,
  ceilingHeightFt: 10,
  elephant: 'present',
// ~~~~~~~~~~~~~~~~~~ Object literal may only specify known properties,
//                    and 'elephant' does not exist in type 'Room'
};
```

While it is odd that there's an `elephant` property, this error doesn't make much sense from a structural typing point of view (Item 4). That constant *is* assignable to the `Room` type, which you can see by introducing an intermediate variable:

```
const obj = {
  numDoors: 1,
```

```
  ceilingHeightFt: 10,
  elephant: 'present',
};
const r: Room = obj;  // OK
```

The type of obj is inferred as `{ numDoors: number; ceilingHeightFt: number; elephant: string }`. Because this type includes a subset of the values in the Room type, it is assignable to Room, and the code passes the type checker (see Item 7).

So what is different about these two examples? In the first you've triggered a process known as "excess property checking," which helps catch an important class of errors that the structural type system would otherwise miss. But this process has its limits, and conflating it with regular assignability checks can make it harder to build an intuition for structural typing. Recognizing excess property checking as a distinct process will help you build a clearer mental model of TypeScript's type system.

As Item 1 explained, TypeScript goes beyond trying to flag code that will throw exceptions at runtime. It also tries to find code that doesn't do what you intend. Here's an example of the latter:

```
interface Options {
  title: string;
  darkMode?: boolean;
}
function createWindow(options: Options) {
  if (options.darkMode) {
    setDarkMode();
  }
  // ...
}
createWindow({
  title: 'Spider Solitaire',
  darkmode: true
// ~~~~~~~~~~~~~ Object literal may only specify known properties, but
//               'darkmode' does not exist in type 'Options'.
//               Did you mean to write 'darkMode'?
});
```

This code doesn't throw any sort of error at runtime. But it's also unlikely to do what you intended for the exact reason that TypeScript says: it should be darkMode (capital M), not darkmode.

A purely structural type checker wouldn't be able to spot this sort of error because the domain of the Options type is incredibly broad: it includes all objects with a title property that's a string and *any other properties*, so long as those don't include a darkMode property set to something other than true or false.

It's easy to forget how expansive TypeScript types can be. Here are a few more values that are assignable to Options:

```
const o1: Options = document;  // OK
const o2: Options = new HTMLAnchorElement;  // OK
```

Both `document` and instances of `HTMLAnchorElement` have `title` properties that are strings, so these assignments are OK. `Options` is a broad type indeed!

Excess property checking tries to rein this in without compromising the fundamentally structural nature of the type system. It does this by disallowing unknown properties specifically on object literals. (It's sometimes called "strict object literal checking" for this reason.) Neither `document` nor `new HTMLAnchorElement` is an object literal, so they did not trigger the checks. But the `{title, darkmode}` object is, so it does:

```
const o: Options = { darkmode: true, title: 'Ski Free' };
                  // ~~~~~~~~ 'darkmode' does not exist in type 'Options'...
```

This explains why using an intermediate variable without a type annotation makes the error go away:

```
const intermediate = { darkmode: true, title: 'Ski Free' };
const o: Options = intermediate;  // OK
```

While the righthand side of the first line is an object literal, the righthand side of the second line (`intermediate`) is not, so excess property checking does not apply, and the error goes away.

Excess property checking does not happen when you use a type assertion:

```
const o = { darkmode: true, title: 'Ski Free' } as Options;  // OK
```

This is a good reason to prefer declarations to assertions (Item 9).

If you don't want this sort of check, you can tell TypeScript to expect additional properties using an index signature:

```
interface Options {
  darkMode?: boolean;
  [otherOptions: string]: unknown;
}
const o: Options = { darkmode: true };  // OK
```

Item 15 discusses when this is and is not an appropriate way to model your data.

A related check happens for "weak" types, which have only optional properties:

```
interface LineChartOptions {
  logscale?: boolean;
  invertedYAxis?: boolean;
  areaChart?: boolean;
}
const opts = { logScale: true };
const o: LineChartOptions = opts;
```

```
   // ~ Type '{ logScale: boolean; }' has no properties in common
   //   with type 'LineChartOptions'
```

From a structural point of view, the `LineChartOptions` type should include almost all objects. For weak types like this, TypeScript adds another check to make sure that the value type and declared type have at least one property in common. Much like excess property checking, this is effective at catching typos and isn't strictly structural. But unlike excess property checking, it happens during all assignability checks involving weak types. Factoring out an intermediate variable doesn't bypass this check.

Excess property checking is an effective way of catching typos and other mistakes in property names that would otherwise be allowed by the structural typing system. It's particularly useful with types like `Options` that contain optional fields. But it is also very limited in scope: it only applies to object literals. Recognize this limitation and distinguish between excess property checking and ordinary type checking. This will help you build a mental model of both.

Factoring out a constant made an error go away here, but it can also introduce an error in other contexts. See Item 26 for examples of this.

Things to Remember

- When you assign an object literal to a variable or pass it as an argument to a function, it undergoes excess property checking.
- Excess property checking is an effective way to find errors, but it is distinct from the usual structural assignability checks done by the TypeScript type checker. Conflating these processes will make it harder for you to build a mental model of assignability.
- Be aware of the limits of excess property checking: introducing an intermediate variable will remove these checks.

Item 12: Apply Types to Entire Function Expressions When Possible

JavaScript (and TypeScript) distinguishes a function *statement* and a function *expression*:

```
function rollDice1(sides: number): number { /* ... */ }  // Statement
const rollDice2 = function(sides: number): number { /* ... */ };  // Expression
const rollDice3 = (sides: number): number => { /* ... */ };  // Also expression
```

An advantage of function expressions in TypeScript is that you can apply a type declaration to the entire function at once, rather than specifying the types of the parameters and return type individually:

```
type DiceRollFn = (sides: number) => number;
const rollDice: DiceRollFn = sides => { /* ... */ };
```

If you mouse over `sides` in your editor, you'll see that TypeScript knows its type is `number`. The function type doesn't provide much value in such a simple example, but the technique does open up a number of possibilities.

One is reducing repetition. If you wanted to write several functions for doing arithmetic on numbers, for instance, you could write them like this:

```
function add(a: number, b: number) { return a + b; }
function sub(a: number, b: number) { return a - b; }
function mul(a: number, b: number) { return a * b; }
function div(a: number, b: number) { return a / b; }
```

or consolidate the repeated function signatures with a single function type:

```
type BinaryFn = (a: number, b: number) => number;
const add: BinaryFn = (a, b) => a + b;
const sub: BinaryFn = (a, b) => a - b;
const mul: BinaryFn = (a, b) => a * b;
const div: BinaryFn = (a, b) => a / b;
```

This has fewer type annotations than before, and they're separated away from the function implementations. This makes the logic more apparent. You've also gained a check that the return type of all the function expressions is `number`.

Libraries often provide types for common function signatures. For example, ReactJS provides a `MouseEventHandler` type that you can apply to an entire function rather than specifying `MouseEvent` as a type for the function's parameter. If you're a library author, consider providing type declarations for common callbacks.

Another place you might want to apply a type to a function expression is to match the signature of some other function. In a web browser, for example, the `fetch` function issues an HTTP request for some resource:

```
const responseP = fetch('/quote?by=Mark+Twain');  // Type is Promise<Response>
```

You extract data from the response via `response.json()` or `response.text()`:

```
async function getQuote() {
  const response = await fetch('/quote?by=Mark+Twain');
  const quote = await response.json();
  return quote;
}
// {
//   "quote": "If you tell the truth, you don't have to remember anything.",
//   "source": "notebook",
//   "date": "1894"
// }
```

(See Item 25 for more on Promises and `async`/`await`.)

There's a bug here: if the request for /quote fails, the response body is likely to contain an explanation like "404 Not Found." This isn't JSON, so `response.json()` will return a rejected Promise with a message about invalid JSON. This obscures the real error, which was a 404.

It's easy to forget that an error response with `fetch` does not result in a rejected Promise. Let's write a `checkedFetch` function to do the status check for us. The type declarations for `fetch` in `lib.dom.d.ts` look like this:

```
declare function fetch(
  input: RequestInfo, init?: RequestInit
): Promise<Response>;
```

So you can write `checkedFetch` like this:

```
async function checkedFetch(input: RequestInfo, init?: RequestInit) {
  const response = await fetch(input, init);
  if (!response.ok) {
    // Converted to a rejected Promise in an async function
    throw new Error('Request failed: ' + response.status);
  }
  return response;
}
```

This works, but it can be written more concisely:

```
const checkedFetch: typeof fetch = async (input, init) => {
  const response = await fetch(input, init);
  if (!response.ok) {
    throw new Error('Request failed: ' + response.status);
  }
  return response;
}
```

We've changed from a function statement to a function expression and applied a type (`typeof fetch`) to the entire function. This allows TypeScript to infer the types of the `input` and `init` parameters.

The type annotation also guarantees that the return type of `checkedFetch` will be the same as that of `fetch`. Had you written `return` instead of `throw`, for example, TypeScript would have caught the mistake:

```
const checkedFetch: typeof fetch = async (input, init) => {
  //  ~~~~~~~~~~~~   Type 'Promise<Response | HTTPError>'
  //                     is not assignable to type 'Promise<Response>'
  //                   Type 'Response | HTTPError' is not assignable
  //                       to type 'Response'
  const response = await fetch(input, init);
  if (!response.ok) {
    return new Error('Request failed: ' + response.status);
  }
```

```
    return response;
  }
```

The same mistake in the first example would likely have led to an error, but in the code that called `checkedFetch`, rather than in the implementation.

In addition to being more concise, typing this entire function expression instead of its parameters has given you better safety. When you're writing a function that has the same type signature as another one, or writing many functions with the same type signature, consider whether you can apply a type declaration to entire functions, rather than repeating types of parameters and return values.

Things to Remember

- Consider applying type annotations to entire function expressions, rather than to their parameters and return type.
- If you're writing the same type signature repeatedly, factor out a function type or look for an existing one. If you're a library author, provide types for common callbacks.
- Use `typeof fn` to match the signature of another function.

Item 13: Know the Differences Between type and interface

If you want to define a named type in TypeScript, you have two options. You can use a type, as shown here:

```
type TState = {
  name: string;
  capital: string;
}
```

or an interface:

```
interface IState {
  name: string;
  capital: string;
}
```

(You could also use a `class`, but that is a JavaScript runtime concept that also introduces a value. See Item 8.)

Which should you use, `type` or `interface`? The line between these two options has become increasingly blurred over the years, to the point that in many situations you can use either. You should be aware of the distinctions that remain between `type` and `interface` and be consistent about which you use in which situation. But you should also know how to write the same types using both, so that you'll be comfortable reading TypeScript that uses either.

> The examples in this item prefix type names with `I` or `T` solely to indicate how they were defined. You should not do this in your code! Prefixing interface types with `I` is common in C#, and this convention made some inroads in the early days of TypeScript. But it is considered bad style today because it's unnecessary, adds little value, and is not consistently followed in the standard libraries.

First, the similarities: the State types are nearly indistinguishable from one another. If you define an `IState` or a `TState` value with an extra property, the errors you get are character-by-character identical:

```
const wyoming: TState = {
  name: 'Wyoming',
  capital: 'Cheyenne',
  population: 500_000
// ~~~~~~~~~~~~~~~~~~ Type ... is not assignable to type 'TState'
//                    Object literal may only specify known properties, and
//                    'population' does not exist in type 'TState'
};
```

You can use an index signature with both `interface` and `type`:

```
type TDict = { [key: string]: string };
interface IDict {
  [key: string]: string;
}
```

You can also define function types with either:

```
type TFn = (x: number) => string;
interface IFn {
  (x: number): string;
}

const toStrT: TFn = x => '' + x;  // OK
const toStrI: IFn = x => '' + x;  // OK
```

The type alias looks more natural for this straightforward function type, but if the type has properties as well, then the declarations start to look more alike:

```
type TFnWithProperties = {
  (x: number): number;
  prop: string;
}
interface IFnWithProperties {
  (x: number): number;
  prop: string;
}
```

You can remember this syntax by reminding yourself that in JavaScript, functions are callable objects.

Both type aliases and interfaces can be generic:

```
type TPair<T> = {
  first: T;
  second: T;
}
interface IPair<T> {
  first: T;
  second: T;
}
```

An `interface` can extend a `type` (with some caveats, explained momentarily), and a `type` can extend an `interface`:

```
interface IStateWithPop extends TState {
  population: number;
}
type TStateWithPop = IState & { population: number; };
```

Again, these types are identical. The caveat is that an `interface` cannot extend a complex type like a union type. If you want to do that, you'll need to use `type` and `&`.

A class can implement either an `interface` or a simple type:

```
class StateT implements TState {
  name: string = '';
  capital: string = '';
}
class StateI implements IState {
  name: string = '';
  capital: string = '';
}
```

Those are the similarities. What about the differences? You've seen one already—there are union `type`s but no union `interface`s:

```
type AorB = 'a' | 'b';
```

Extending union types can be useful. If you have separate types for `Input` and `Output` variables and a mapping from name to variable:

```
type Input = { /* ... */ };
type Output = { /* ... */ };
interface VariableMap {
  [name: string]: Input | Output;
}
```

then you might want a type that attaches the name to the variable. This would be:

```
type NamedVariable = (Input | Output) & { name: string };
```

This type cannot be expressed with `interface`. A `type` is, in general, more capable than an `interface`. It can be a union, and it can also take advantage of more advanced features like mapped or conditional types.

It can also more easily express tuple and array types:

```
type Pair = [number, number];
type StringList = string[];
type NamedNums = [string, ...number[]];
```

You can express something *like* a tuple using `interface`:

```
interface Tuple {
  0: number;
  1: number;
  length: 2;
}
const t: Tuple = [10, 20];  // OK
```

But this is awkward and drops all the tuple methods like `concat`. Better to use a `type`. For more on the problems of numeric indices, see Item 16.

An `interface` does have some abilities that a `type` doesn't, however. One of these is that an `interface` can be *augmented*. Going back to the `State` example, you could have added a `population` field in another way:

```
interface IState {
  name: string;
  capital: string;
}
interface IState {
  population: number;
}
const wyoming: IState = {
  name: 'Wyoming',
  capital: 'Cheyenne',
  population: 500_000
};  // OK
```

This is known as "declaration merging," and it's quite surprising if you've never seen it before. This is primarily used with type declaration files (Chapter 6), and if you're writing one, you should follow the norms and use `interface` to support it. The idea is that there may be gaps in your type declarations that users need to fill, and this is how they do it.

TypeScript uses merging to get different types for the different versions of JavaScript's standard library. The `Array` interface, for example, is defined in *lib.es5.d.ts*. By default this is all you get. But if you add `ES2015` to the `lib` entry of your *tsconfig.json*, TypeScript will also include *lib.es2015.d.ts*. This includes another `Array` interface with additional methods like `find` that were added in ES2015. They get added to the other

`Array` interface via merging. The net effect is that you get a single `Array` type with exactly the right methods.

Merging is supported in regular code as well as declarations, and you should be aware of the possibility. If it's essential that no one ever augment your type, then use `type`.

Returning to the question at the start of the item, should you use `type` or `interface`? For complex types, you have no choice: you need to use a type alias. But what about the simpler object types that can be represented either way? To answer this question, you should consider consistency and augmentation. Are you working in a codebase that consistently uses `interface`? Then stick with `interface`. Does it use `type`? Then use `type`.

For projects without an established style, you should think about augmentation. Are you publishing type declarations for an API? Then it might be helpful for your users to be able to be able to merge in new fields via an `interface` when the API changes. So use `interface`. But for a type that's used internally in your project, declaration merging is likely to be a mistake. So prefer `type`.

Things to Remember

- Understand the differences and similarities between `type` and `interface`.
- Know how to write the same types using either syntax.
- In deciding which to use in your project, consider the established style and whether augmentation might be beneficial.

Item 14: Use Type Operations and Generics to Avoid Repeating Yourself

This script prints the dimensions, surface areas, and volumes of a few cylinders:

```
console.log('Cylinder 1 x 1 ',
  'Surface area:', 6.283185 * 1 * 1 + 6.283185 * 1 * 1,
  'Volume:', 3.14159 * 1 * 1 * 1);
console.log('Cylinder 1 x 2 ',
  'Surface area:', 6.283185 * 1 * 1 + 6.283185 * 2 * 1,
  'Volume:', 3.14159 * 1 * 2 * 1);
console.log('Cylinder 2 x 1 ',
  'Surface area:', 6.283185 * 2 * 1 + 6.283185 * 2 * 1,
  'Volume:', 3.14159 * 2 * 2 * 1);
```

Is this code uncomfortable to look at? It should be. It's extremely repetitive, as though the same line was copied and pasted, then modified. It repeats both values and constants. This has allowed an error to creep in (did you spot it?). Much better would be to factor out some functions, a constant, and a loop:

```
const surfaceArea = (r, h) => 2 * Math.PI * r * (r + h);
const volume = (r, h) => Math.PI * r * r * h;
for (const [r, h] of [[1, 1], [1, 2], [2, 1]]) {
  console.log(
    `Cylinder ${r} x ${h}`,
    `Surface area: ${surfaceArea(r, h)}`,
    `Volume: ${volume(r, h)}`);
}
```

This is the DRY principle: don't repeat yourself. It's the closest thing to universal advice that you'll find in software development. Yet developers who assiduously avoid repetition in code may not think twice about it in types:

```
interface Person {
  firstName: string;
  lastName: string;
}

interface PersonWithBirthDate {
  firstName: string;
  lastName: string;
  birth: Date;
}
```

Duplication in types has many of the same problems as duplication in code. What if you decide to add an optional `middleName` field to `Person`? Now `Person` and `PersonWithBirthDate` have diverged.

One reason that duplication is more common in types is that the mechanisms for factoring out shared patterns are less familiar than they are with code: what's the type system equivalent of factoring out a helper function? By learning how to map between types, you can bring the benefits of DRY to your type definitions.

The simplest way to reduce repetition is by naming your types. Rather than writing a distance function this way:

```
function distance(a: {x: number, y: number}, b: {x: number, y: number}) {
  return Math.sqrt(Math.pow(a.x - b.x, 2) + Math.pow(a.y - b.y, 2));
}
```

create a name for the type and use that:

```
interface Point2D {
  x: number;
  y: number;
}
function distance(a: Point2D, b: Point2D) { /* ... */ }
```

This is the type system equivalent of factoring out a constant instead of writing it repeatedly. Duplicated types aren't always so easy to spot. Sometimes they can be obscured by syntax. If several functions share the same type signature, for instance:

```
function get(url: string, opts: Options): Promise<Response> { /* ... */ }
function post(url: string, opts: Options): Promise<Response> { /* ... */ }
```

Then you can factor out a named type for this signature:

```
type HTTPFunction = (url: string, opts: Options) => Promise<Response>;
const get: HTTPFunction = (url, opts) => { /* ... */ };
const post: HTTPFunction = (url, opts) => { /* ... */ };
```

For more on this, see Item 12.

What about the `Person`/`PersonWithBirthDate` example? You can eliminate the repetition by making one interface extend the other:

```
interface Person {
  firstName: string;
  lastName: string;
}

interface PersonWithBirthDate extends Person {
  birth: Date;
}
```

Now you only need to write the additional fields. If the two interfaces share a subset of their fields, then you can factor out a base class with just these common fields. Continuing the analogy with code duplication, this is akin to writing `PI` and `2*PI` instead of `3.141593` and `6.283185`.

You can also use the intersection operator (`&`) to extend an existing type, though this is less common:

```
type PersonWithBirthDate = Person & { birth: Date };
```

This technique is most useful when you want to add some additional properties to a union type (which you cannot `extend`). For more on this, see Item 13.

You can also go the other direction. What if you have a type, `State`, which represents the state of an entire application, and another, `TopNavState`, which represents just a part?

```
interface State {
  userId: string;
  pageTitle: string;
  recentFiles: string[];
  pageContents: string;
}
interface TopNavState {
  userId: string;
  pageTitle: string;
  recentFiles: string[];
}
```

Rather than building up `State` by extending `TopNavState`, you'd like to define `TopNavState` as a subset of the fields in `State`. This way you can keep a single interface defining the state for the entire app.

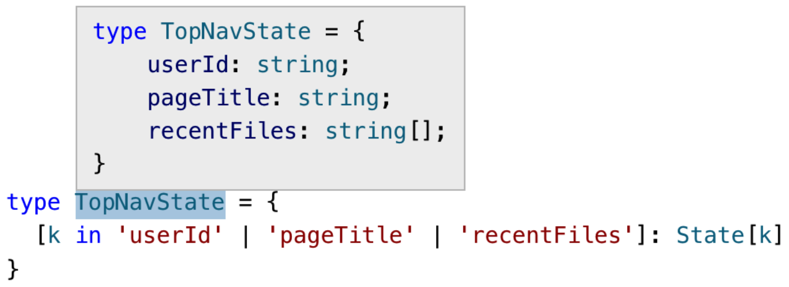
You can remove duplication in the types of the properties by indexing into `State`:

```
type TopNavState = {
  userId: State['userId'];
  pageTitle: State['pageTitle'];
  recentFiles: State['recentFiles'];
};
```

While it's longer, this *is* progress: a change in the type of `pageTitle` in `State` will get reflected in `TopNavState`. But it's still repetitive. You can do better with a *mapped type*:

```
type TopNavState = {
  [k in 'userId' | 'pageTitle' | 'recentFiles']: State[k]
};
```

Mousing over `TopNavState` shows that this definition is, in fact, exactly the same as the previous one (see Figure 2-10).



```
type TopNavState = {
    userId: string;
    pageTitle: string;
    recentFiles: string[];
}
type TopNavState = {
  [k in 'userId' | 'pageTitle' | 'recentFiles']: State[k]
}
```

Figure 2-10. Showing the expanded version of a mapped type in your text editor. This is the same as the initial definition, but with less duplication.

Mapped types are the type system equivalent of looping over the fields in an array. This particular pattern is so common that it's part of the standard library, where it's called `Pick`:

```
type Pick<T, K> = { [k in K]: T[k] };
```

(This definition isn't *quite* complete, as you will see.) You use it like this:

```
type TopNavState = Pick<State, 'userId' | 'pageTitle' | 'recentFiles'>;
```

Pick is an example of a *generic type*. Continuing the analogy to removing code duplication, using Pick is the equivalent of calling a function. Pick takes two types, T and K, and returns a third, much as a function might take two values and return a third.

Another form of duplication can arise with tagged unions. What if you want a type for just the tag?

```
interface SaveAction {
  type: 'save';
  // ...
}
interface LoadAction {
  type: 'load';
  // ...
}
type Action = SaveAction | LoadAction;
type ActionType = 'save' | 'load';  // Repeated types!
```

You can define ActionType without repeating yourself by indexing into the Action union:

```
type ActionType = Action['type'];  // Type is "save" | "load"
```

As you add more types to the Action union, ActionType will incorporate them automatically. This type is distinct from what you'd get using Pick, which would give you an interface with a type property:

```
type ActionRec = Pick<Action, 'type'>;  // {type: "save" | "load"}
```

If you're defining a class which can be initialized and later updated, the type for the parameter to the update method will optionally include most of the same parameters as the constructor:

```
interface Options {
  width: number;
  height: number;
  color: string;
  label: string;
}
interface OptionsUpdate {
  width?: number;
  height?: number;
  color?: string;
  label?: string;
}
class UIWidget {
  constructor(init: Options) { /* ... */ }
  update(options: OptionsUpdate) { /* ... */ }
}
```

You can construct OptionsUpdate from Options using a mapped type and keyof:

```
type OptionsUpdate = {[k in keyof Options]?: Options[k]};
```

`keyof` takes a type and gives you a union of the types of its keys:

```
type OptionsKeys = keyof Options;
// Type is "width" | "height" | "color" | "label"
```

The mapped type (`[k in keyof Options]`) iterates over these and looks up the corresponding value type in `Options`. The `?` makes each property optional. This pattern is also extremely common and is enshrined in the standard library as `Partial`:

```
class UIWidget {
  constructor(init: Options) { /* ... */ }
  update(options: Partial<Options>) { /* ... */ }
}
```

You may also find yourself wanting to define a type that matches the shape of a *value*:

```
const INIT_OPTIONS = {
  width: 640,
  height: 480,
  color: '#00FF00',
  label: 'VGA',
};
interface Options {
  width: number;
  height: number;
  color: string;
  label: string;
}
```

You can do so with `typeof`:

```
type Options = typeof INIT_OPTIONS;
```

This intentionally evokes JavaScript's runtime `typeof` operator, but it operates at the level of TypeScript types and is much more precise. For more on `typeof`, see Item 8. Be careful about deriving types from values, however. It's usually better to define types first and declare that values are assignable to them. This makes your types more explicit and less subject to the vagaries of widening (Item 21).

Similarly, you may want to create a named type for the inferred return value of a function or method:

```
function getUserInfo(userId: string) {
  // ...
  return {
    userId,
    name,
    age,
    height,
    weight,
    favoriteColor,
  };
}
// Return type inferred as { userId: string; name: string; age: number, ... }
```

Doing this directly requires conditional types (see Item 50). But, as we've seen before, the standard library defines generic types for common patterns like this one. In this case the `ReturnType` generic does exactly what you want:

```
type UserInfo = ReturnType<typeof getUserInfo>;
```

Note that `ReturnType` operates on `typeof getUserInfo`, the function's *type*, rather than `getUserInfo`, the function's *value*. As with `typeof`, use this technique judiciously. Don't get mixed up about your source of truth.

Generic types are the equivalent of functions for types. And functions are the key to DRY for logic. So it should come as no surprise that generics are the key to DRY for types. But there's a missing piece to this analogy. You use the type system to constrain the values you can map with a function: you add numbers, not objects; you find the area of shapes, not database records. How do you constrain the parameters in a generic type?

You do so with `extends`. You can declare that any generic parameter `extends` a type. For example:

```
interface Name {
  first: string;
  last: string;
}
type DancingDuo<T extends Name> = [T, T];

const couple1: DancingDuo<Name> = [
  {first: 'Fred', last: 'Astaire'},
  {first: 'Ginger', last: 'Rogers'}
];  // OK
const couple2: DancingDuo<{first: string}> = [
                       // ~~~~~~~~~~~~~~~
                       // Property 'last' is missing in type
                       // '{ first: string; }' but required in type 'Name'
  {first: 'Sonny'},
  {first: 'Cher'}
];
```

`{first: string}` does not extend `Name`, hence the error.

At the moment, TypeScript always requires you to write out the generic parameter in a declaration. Writing `DancingDuo` instead of `DancingDuo<Name>` won't cut it. If you want TypeScript to infer the type of the generic parameter, you can use a carefully typed identity function:

```
const dancingDuo = <T extends Name>(x: DancingDuo<T>) => x;
const couple1 = dancingDuo([
  {first: 'Fred', last: 'Astaire'},
  {first: 'Ginger', last: 'Rogers'}
]);
const couple2 = dancingDuo([
  {first: 'Bono'},
// ~~~~~~~~~~~~~~
  {first: 'Prince'}
// ~~~~~~~~~~~~~~~~
//     Property 'last' is missing in type
//     '{ first: string; }' but required in type 'Name'
]);
```

For a particularly useful variation on this, see `inferringPick` in Item 26.

You can use `extends` to complete the definition of `Pick` from earlier. If you run the original version through the type checker, you get an error:

```
type Pick<T, K> = {
  [k in K]: T[k]
     // ~ Type 'K' is not assignable to type 'string | number | symbol'
};
```

`K` is unconstrained in this type and is clearly too broad: it needs to be something that can be used as an index, namely, `string | number | symbol`. But you can get narrower than that—`K` should really be some subset of the keys of `T`, namely, `keyof T`:

```
type Pick<T, K extends keyof T> = {
  [k in K]: T[k]
};  // OK
```

Thinking of types as sets of values (Item 7), it helps to read "extends" as "subset of" here.

As you work with increasingly abstract types, try not to lose sight of the goal: accepting valid programs and rejecting invalid ones. In this case, the upshot of the constraint is that passing `Pick` the wrong key will produce an error:

```
type FirstLast = Pick<Name, 'first' | 'last'>;  // OK
type FirstMiddle = Pick<Name, 'first' | 'middle'>;
                           // ~~~~~~~~~~~~~~~~~~
                           // Type '"middle"' is not assignable
                           // to type '"first" | "last"'
```

Repetition and copy/paste coding are just as bad in type space as they are in value space. The constructs you use to avoid repetition in type space may be less familiar than those used for program logic, but they are worth the effort to learn. Don't repeat yourself!

Things to Remember

- The DRY (don't repeat yourself) principle applies to types as much as it applies to logic.
- Name types rather than repeating them. Use `extends` to avoid repeating fields in interfaces.
- Build an understanding of the tools provided by TypeScript to map between types. These include `keyof`, `typeof`, indexing, and mapped types.
- Generic types are the equivalent of functions for types. Use them to map between types instead of repeating types. Use `extends` to constrain generic types.
- Familiarize yourself with generic types defined in the standard library such as `Pick`, `Partial`, and `ReturnType`.

Item 15: Use Index Signatures for Dynamic Data

One of the best features of JavaScript is its convenient syntax for creating objects:

```
const rocket = {
  name: 'Falcon 9',
  variant: 'Block 5',
  thrust: '7,607 kN',
};
```

Objects in JavaScript map string keys to values of any type. TypeScript lets you represent flexible mappings like this by specifying an *index signature* on the type:

```
type Rocket = {[property: string]: string};
const rocket: Rocket = {
  name: 'Falcon 9',
  variant: 'v1.0',
  thrust: '4,940 kN',
};  // OK
```

The `[property: string]: string` is the index signature. It specifies three things:

A name for the keys
: This is purely for documentation; it is not used by the type checker in any way.

A type for the key
: This needs to be some combination of `string`, `number`, or `symbol`, but generally you just want to use `string` (see Item 16).

A type for the values
: This can be anything.

While this does type check, it has a few downsides:

- It allows any keys, including incorrect ones. Had you written `Name` instead of `name`, it would have still been a valid `Rocket` type.
- It doesn't require any specific keys to be present. `{}` is also a valid `Rocket`.
- It cannot have distinct types for different keys. For example, `thrust` should probably be a `number`, not a `string`.
- TypeScript's language services can't help you with types like this. As you're typing `name:`, there's no autocomplete because the key could be anything.

In short, index signatures are not very precise. There are almost always better alternatives to them. In this case, `Rocket` should clearly be an `interface`:

```
interface Rocket {
  name: string;
  variant: string;
  thrust_kN: number;
}
const falconHeavy: Rocket = {
  name: 'Falcon Heavy',
  variant: 'v1',
  thrust_kN: 15_200
};
```

Now `thrust_kN` is a `number` and TypeScript will check for the presence of all required fields. All the great language services that TypeScript provides are available: autocomplete, jump to definition, rename—and they all work.

What should you use index signatures for? The canonical case is truly dynamic data. This might come from a CSV file, for instance, where you have a header row and want to represent data rows as objects mapping column names to values:

```
function parseCSV(input: string): {[columnName: string]: string}[] {
  const lines = input.split('\n');
  const [header, ...rows] = lines;
  return rows.map(rowStr => {
    const row: {[columnName: string]: string} = {};
    rowStr.split(',').forEach((cell, i) => {
      row[header[i]] = cell;
    });
    return row;
    });
  }
```

There's no way to know in advance what the column names are in such a general setting. So an index signature is appropriate. If the user of `parseCSV` knows more about what the columns are in a particular context, they may want to use an assertion to get a more specific type:

```
interface ProductRow {
  productId: string;
  name: string;
  price: string;
}

declare let csvData: string;
const products = parseCSV(csvData) as unknown as ProductRow[];
```

Of course, there's no guarantee that the columns at runtime will actually match your expectation. If this is something you're concerned about, you can add `undefined` to the value type:

```
function safeParseCSV(
  input: string
): {[columnName: string]: string | undefined}[] {
  return parseCSV(input);
}
```

Now every access requires a check:

```
const rows = parseCSV(csvData);
const prices: {[produt: string]: number} = {};
for (const row of rows) {
  prices[row.productId] = Number(row.price);
}

const safeRows = safeParseCSV(csvData);
for (const row of safeRows) {
  prices[row.productId] = Number(row.price);
      // ~~~~~~~~~~~~~ Type 'undefined' cannot be used as an index type
}
```

Of course, this may make the type less convenient to work with. Use your judgment.

If your type has a limited set of possible fields, don't model this with an index signature. For instance, if you know your data will have keys like A, B, C, D, but you don't know how many of them there will be, you could model the type either with optional fields or a union:

```
interface Row1 { [column: string]: number }  // Too broad
interface Row2 { a: number; b?: number; c?: number; d?: number }  // Better
type Row3 =
    | { a: number; }
    | { a: number; b: number; }
```

```
      | { a: number; b: number; c: number;  }
      | { a: number; b: number; c: number; d: number };
```

The last form is the most precise, but it may be less convenient to work with.

If the problem with using an index signature is that `string` is too broad, then there are a few alternatives.

One is using `Record`. This is a generic type that gives you more flexibility in the key type. In particular, you can pass in subsets of `string`:

```
type Vec3D = Record<'x' | 'y' | 'z', number>;
// Type Vec3D = {
//   x: number;
//   y: number;
//   z: number;
// }
```

Another is using a mapped type. This gives you the possibility of using different types for different keys:

```
type Vec3D = {[k in 'x' | 'y' | 'z']: number};
// Same as above
type ABC = {[k in 'a' | 'b' | 'c']: k extends 'b' ? string : number};
// Type ABC = {
//   a: number;
//   b: string;
//   c: number;
// }
```

Things to Remember

- Use index signatures when the properties of an object cannot be known until runtime—for example, if you're loading them from a CSV file.
- Consider adding `undefined` to the value type of an index signature for safer access.
- Prefer more precise types to index signatures when possible: `interface`s, `Record`s, or mapped types.

Item 16: Prefer Arrays, Tuples, and ArrayLike to number Index Signatures

JavaScript is a famously quirky language. Some of the most notorious quirks involve implicit type coercions:

```
> "0" == 0
true
```

but these can usually be avoided by using === and !== instead of their more coercive cousins.

JavaScript's object model also has its quirks, and these are more important to understand because some of them are modeled by TypeScript's type system. You've already seen one such quirk in Item 10, which discussed object wrapper types. This item discusses another.

What is an object? In JavaScript it's a collection of key/value pairs. The keys are usually strings (in ES2015 and later they can also be symbols). The values can be anything.

This is more restrictive than what you find in many other languages. JavaScript does not have a notion of "hashable" objects like you find in Python or Java. If you try to use a more complex object as a key, it is converted into a string by calling its `toString` method:

```
> x = {}
{}
> x[[1, 2, 3]] = 2
2
> x
{ '1,2,3': 1 }
```

In particular, *numbers* cannot be used as keys. If you try to use a number as a property name, the JavaScript runtime will convert it to a string:

```
> { 1: 2, 3: 4}
{ '1': 2, '3': 4 }
```

So what are arrays, then? They are certainly objects:

```
> typeof []
'object'
```

And yet it's quite normal to use numeric indices with them:

```
> x = [1, 2, 3]
[ 1, 2, 3 ]
> x[0]
1
```

Are these being converted into strings? In one of the oddest quirks of all, the answer is "yes." You can also access the elements of an array using string keys:

```
> x['1']
2
```

If you use `Object.keys` to list the keys of an array, you get strings back:

```
> Object.keys(x)
[ '0', '1', '2' ]
```

TypeScript attempts to bring some sanity to this by allowing numeric keys and distinguishing between these and strings. If you dig into the type declarations for `Array` (Item 6), you'll find this in *lib.es5.d.ts*:

```
interface Array<T> {
  // ...
  [n: number]: T;
}
```

This is purely a fiction—string keys are accepted at runtime as the ECMAScript standard dictates that they must—but it is a helpful one that can catch mistakes:

```
const xs = [1, 2, 3];
const x0 = xs[0];  // OK
const x1 = xs['1'];
           // ~~~ Element implicitly has an 'any' type
           //      because index expression is not of type 'number'

function get<T>(array: T[], k: string): T {
  return array[k];
            // ~ Element implicitly has an 'any' type
            //   because index expression is not of type 'number'
}
```

While this fiction is helpful, it's important to remember that it is just a fiction. Like all aspects of TypeScript's type system, it is erased at runtime (Item 3). This means that constructs like `Object.keys` still return strings:

```
const keys = Object.keys(xs);  // Type is string[]
for (const key in xs) {
  key;  // Type is string
  const x = xs[key];  // Type is number
}
```

That this last access works is somewhat surprising since `string` is not assignable to `number`. It's best thought of as a pragmatic concession to this style of iterating over arrays, which is common in JavaScript. That's not to say that this is a good way to loop over an array. If you don't care about the index, you can use for-of:

```
for (const x of xs) {
  x;  // Type is number
}
```

If you do care about the index, you can use `Array.prototype.forEach`, which gives it to you as a `number`:

```
xs.forEach((x, i) => {
  i;  // Type is number
  x;  // Type is number
});
```

If you need to break out of the loop early, you're best off using a C-style `for(;;)` loop:

```
for (let i = 0; i < xs.length; i++) {
  const x = xs[i];
  if (x < 0) break;
}
```

If the types don't convince you, perhaps the performance will: in most browsers and JavaScript engines, for-in loops over arrays are several orders of magnitude slower than for-of or a C-style for loop.

The general pattern here is that a `number` index signature means that what you put in has to be a `number` (with the notable exception of for-in loops), but what you get out is a `string`.

If this sounds confusing, it's because it is! As a general rule, there's not much reason to use `number` as the index signature of a type rather than `string`. If you want to specify something that will be indexed using numbers, you probably want to use an Array or tuple type instead. Using `number` as an index type can create the misconception that numeric properties are a thing in JavaScript, either for yourself or for readers of your code.

If you object to accepting an Array type because they have many other properties (from their prototype) that you might not use, such as `push` and `concat`, then that's good—you're thinking structurally! (If you need a refresher on this, refer to Item 4.) If you truly want to accept tuples of any length or any array-like construct, TypeScript has an `ArrayLike` type you can use:

```
function checkedAccess<T>(xs: ArrayLike<T>, i: number): T {
  if (i < xs.length) {
    return xs[i];
  }
  throw new Error(`Attempt to access ${i} which is past end of array.`)
}
```

This has just a `length` and numeric index signature. In the rare cases that this is what you want, you should use it instead. But remember that the keys are still really strings!

```
const tupleLike: ArrayLike<string> = {
  '0': 'A',
  '1': 'B',
  length: 2,
};  // OK
```

Things to Remember

- Understand that arrays are objects, so their keys are strings, not numbers. `number` as an index signature is a purely TypeScript construct which is designed to help catch bugs.

- Prefer `Array`, tuple, or `ArrayLike` types to using `number` in an index signature yourself.

Item 17: Use readonly to Avoid Errors Associated with Mutation

Here's some code to print the triangular numbers (1, 1+2, 1+2+3, etc.):

```
function printTriangles(n: number) {
  const nums = [];
  for (let i = 0; i < n; i++) {
    nums.push(i);
    console.log(arraySum(nums));
  }
}
```

This code looks straightforward. But here's what happens when you run it:

```
> printTriangles(5)
0
1
2
3
4
```

The problem is that you've made an assumption about `arraySum`, namely, that it doesn't modify `nums`. But here's my implementation:

```
function arraySum(arr: number[]) {
  let sum = 0, num;
  while ((num = arr.pop()) !== undefined) {
    sum += num;
  }
  return sum;
}
```

This function does calculate the sum of the numbers in the array. But it also has the side effect of emptying the array! TypeScript is fine with this, because JavaScript arrays are mutable.

It would be nice to have some assurances that `arraySum` does not modify the array. This is what the `readonly` type modifier does:

```
function arraySum(arr: readonly number[]) {
  let sum = 0, num;
  while ((num = arr.pop()) !== undefined) {
                   // ~~~ 'pop' does not exist on type 'readonly number[]'
    sum += num;
  }
```

```
  return sum;
}
```

This error message is worth digging into. `readonly number[]` is a *type*, and it is distinct from `number[]` in a few ways:

- You can read from its elements, but you can't write to them.
- You can read its `length`, but you can't set it (which would mutate the array).
- You can't call `pop` or other methods that mutate the array.

Because `number[]` is strictly more capable than `readonly number[]`, it follows that `number[]` is a subtype of `readonly number[]`. (It's easy to get this backwards—remember Item 7!) So you can assign a mutable array to a `readonly` array, but not vice versa:

```
const a: number[] = [1, 2, 3];
const b: readonly number[] = a;
const c: number[] = b;
   // ~ Type 'readonly number[]' is 'readonly' and cannot be
   //   assigned to the mutable type 'number[]'
```

This makes sense: the `readonly` modifier wouldn't be much use if you could get rid of it without even a type assertion.

When you declare a parameter `readonly`, a few things happen:

- TypeScript checks that the parameter isn't mutated in the function body.
- Callers are assured that your function doesn't mutate the parameter.
- Callers may pass your function a `readonly` array.

There is often an assumption in JavaScript (and TypeScript) that functions don't mutate their parameters unless explicitly noted. But as we'll see time and again in this book (particularly Items 30 and 31), these sorts of implicit understandings can lead to trouble with type checking. Better to make them explicit, both for human readers and for `tsc`.

The fix for `arraySum` is simple: don't mutate the array!

```
function arraySum(arr: readonly number[]) {
  let sum = 0;
  for (const num of arr) {
    sum += num;
  }
  return sum;
}
```

Now `printTriangles` does what you expect:

```
> printTriangles(5)
0
1
3
6
10
```

If your function does not mutate its parameters, then you should declare them `readonly`. There's relatively little downside: users will be able to call them with a broader set of types (Item 29), and inadvertent mutations will be caught.

One downside is that you may need to call functions that haven't marked their parameters `readonly`. If these don't mutate their parameters and are in your control, make them `readonly`! `readonly` tends to be contagious: once you mark one function with `readonly`, you'll also need to mark all the functions that it calls. This is a good thing since it leads to clearer contracts and better type safety. But if you're calling a function in another library, you may not be able to change its type declarations, and you may have to resort to a type assertion (`param as number[]`).

`readonly` can also be used to catch a whole class of mutation errors involving local variables. Imagine you're writing a tool to process a novel. You get a sequence of lines and would like to collect them into paragraphs, which are separated by blanks:

```
Frankenstein; or, The Modern Prometheus
by Mary Shelley

You will rejoice to hear that no disaster has accompanied the commencement
of an enterprise which you have regarded with such evil forebodings. I arrived
here yesterday, and my first task is to assure my dear sister of my welfare and
increasing confidence in the success of my undertaking.

I am already far north of London, and as I walk in the streets of Petersburgh,
I feel a cold northern breeze play upon my cheeks, which braces my nerves and
fills me with delight.
```

Here's an attempt:[1]

```
function parseTaggedText(lines: string[]): string[][] {
  const paragraphs: string[][] = [];
  const currPara: string[] = [];

  const addParagraph = () => {
    if (currPara.length) {
      paragraphs.push(currPara);
      currPara.length = 0;  // Clear the lines
    }
  };
```

1 In practice you might just write `lines.join('\n').split(/\n\n+/)`, but bear with me.

```
  for (const line of lines) {
    if (!line) {
      addParagraph();
    } else {
      currPara.push(line);
    }
  }
  addParagraph();
  return paragraphs;
}
```

When you run this on the example at the beginning of the item, here's what you get:

```
[ [], [], [] ]
```

Well that went horribly wrong!

The problem with this code is a toxic combination of aliasing (Item 24) and mutation. The aliasing happens on this line:

```
paragraphs.push(currPara);
```

Rather than pushing the contents of `currPara`, this pushes a reference to the array. When you push a new value to `currPara` or clear it, this change is also reflected in the entries in `paragraphs` because they point to the same object.

In other words, the net effect of this code:

```
paragraphs.push(currPara);
currPara.length = 0;  // Clear lines
```

is that you push a new paragraph onto `paragraphs` and then immediately clear it.

The problem is that setting `currPara.length` and calling `currPara.push` both mutate the `currPara` array. You can disallow this behavior by declaring it to be `readonly`. This immediately surfaces a few errors in the implementation:

```
function parseTaggedText(lines: string[]): string[][] {
  const currPara: readonly string[] = [];
  const paragraphs: string[][] = [];

  const addParagraph = () => {
    if (currPara.length) {
      paragraphs.push(
        currPara
     // ~~~~~~~~ Type 'readonly string[]' is 'readonly' and
     //          cannot be assigned to the mutable type 'string[]'
      );
      currPara.length = 0;  // Clear lines
            // ~~~~~~ Cannot assign to 'length' because it is a read-only
            // property
    }
  };
```

```
  for (const line of lines) {
    if (!line) {
      addParagraph();
    } else {
      currPara.push(line);
            // ~~~~ Property 'push' does not exist on type 'readonly string[]'
    }
  }
  addParagraph();
  return paragraphs;
}
```

You can fix two of the errors by declaring `currPara` with `let` and using nonmutating methods:

```
let currPara: readonly string[] = [];
// ...
currPara = [];  // Clear lines
// ...
currPara = currPara.concat([line]);
```

Unlike `push`, `concat` returns a new array, leaving the original unmodified. By changing the declaration from `const` to `let` and adding `readonly`, you've traded one sort of mutability for another. The `currPara` variable is now free to change which array it points to, but those arrays themselves are not allowed to change.

This leaves the error about `paragraphs`. You have three options for fixing this.

First, you could make a copy of `currPara`:

```
paragraphs.push([...currPara]);
```

This fixes the error because, while `currPara` remains `readonly`, you're free to mutate the copy however you like.

Second, you could change `paragraphs` (and the return type of the function) to be an array of `readonly string[]`:

```
const paragraphs: (readonly string[])[] = [];
```

(The grouping is relevant here: `readonly string[][]` would be a `readonly` array of mutable arrays, rather than a mutable array of `readonly` arrays.)

This works, but it seems a bit rude to users of `parseTaggedText`. Why do you care what they do with the paragraphs after the function returns?

Third, you could use an assertion to remove the `readonly`-ness of the array:

```
paragraphs.push(currPara as string[]);
```

Since you're assigning `currPara` to a new array in the very next statement, this doesn't seem like the most offensive assertion.

An important caveat to `readonly` is that it is *shallow*. You saw this with `readonly string[][]` earlier. If you have a `readonly` array of objects, the objects themselves are not `readonly`:

```
const dates: readonly Date[] = [new Date()];
dates.push(new Date());
   // ~~~~ Property 'push' does not exist on type 'readonly Date[]'
dates[0].setFullYear(2037);  // OK
```

Similar considerations apply to `readonly`'s cousin for objects, the `Readonly` generic:

```
interface Outer {
  inner: {
    x: number;
  }
}
const o: Readonly<Outer> = { inner: { x: 0 }};
o.inner = { x: 1 };
// ~~~~ Cannot assign to 'inner' because it is a read-only property
o.inner.x = 1;  // OK
```

You can create a type alias and then inspect it in your editor to see exactly what's happening:

```
type T = Readonly<Outer>;
// Type T = {
//   readonly inner: {
//     x: number;
//   };
// }
```

The important thing to note is the `readonly` modifier on `inner` but not on `x`. There is no built-in support for deep readonly types at the time of this writing, but it is possible to create a generic to do this. Getting this right is tricky, so I recommend using a library rather than rolling your own. The `DeepReadonly` generic in `ts-essentials` is one implementation.

You can also write `readonly` on an index signature. This has the effect of preventing writes but allowing reads:

```
let obj: {readonly [k: string]: number} = {};
// Or Readonly<{[k: string]: number}
obj.hi = 45;
//  ~~ Index signature in type ... only permits reading
obj = {...obj, hi: 12};  // OK
obj = {...obj, bye: 34};  // OK
```

This can prevent issues with aliasing and mutation involving objects rather than arrays.

Things to Remember

- If your function does not modify its parameters then declare them `readonly`. This makes its contract clearer and prevents inadvertent mutations in its implementation.
- Use `readonly` to prevent errors with mutation and to find the places in your code where mutations occur.
- Understand the difference between `const` and `readonly`.
- Understand that `readonly` is shallow.

Item 18: Use Mapped Types to Keep Values in Sync

Suppose you're writing a UI component for drawing scatter plots. It has a few different types of properties that control its display and behavior:

```
interface ScatterProps {
  // The data
  xs: number[];
  ys: number[];

  // Display
  xRange: [number, number];
  yRange: [number, number];
  color: string;

  // Events
  onClick: (x: number, y: number, index: number) => void;
}
```

To avoid unnecessary work, you'd like to redraw the chart only when you need to. Changing data or display properties will require a redraw, but changing the event handler will not. This sort of optimization is common in React components, where an event handler Prop might be set to a new arrow function on every render.[2]

Here's one way you might implement this optimization:

```
function shouldUpdate(
  oldProps: ScatterProps,
  newProps: ScatterProps
) {
  let k: keyof ScatterProps;
  for (k in oldProps) {
    if (oldProps[k] !== newProps[k]) {
```

2 React's `useCallback` hook is another technique to avoid creating new functions on every render.

```
      if (k !== 'onClick') return true;
    }
  }
  return false;
}
```

(See Item 54 for an explanation of the `keyof` declaration in this loop.)

What happens when you or a coworker add a new property? The `shouldUpdate` function will redraw the chart whenever it changes. You might call this the conservative or "fail closed" approach. The upside is that the chart will always look right. The downside is that it might be drawn too often.

A "fail open" approach might look like this:

```
function shouldUpdate(
  oldProps: ScatterProps,
  newProps: ScatterProps
) {
  return (
    oldProps.xs !== newProps.xs ||
    oldProps.ys !== newProps.ys ||
    oldProps.xRange !== newProps.xRange ||
    oldProps.yRange !== newProps.yRange ||
    oldProps.color !== newProps.color
    // (no check for onClick)
  );
}
```

With this approach there won't be any unnecessary redraws, but there might be some *necessary* draws that get dropped. This violates the "first, do no harm" principle of optimization and so is less common.

Neither approach is ideal. What you'd really like is to force your coworker or future self to make a decision when adding the new property. You might try adding a comment:

```
interface ScatterProps {
  xs: number[];
  ys: number[];
  // ...
  onClick: (x: number, y: number, index: number) => void;

  // Note: if you add a property here, update shouldUpdate!
}
```

But do you really expect this to work? It would be better if the type checker could enforce this for you.

If you set it up the right way, it can. The key is to use a mapped type and an object:

```
const REQUIRES_UPDATE: {[k in keyof ScatterProps]: boolean} = {
  xs: true,
  ys: true,
  xRange: true,
  yRange: true,
  color: true,
  onClick: false,
};

function shouldUpdate(
  oldProps: ScatterProps,
  newProps: ScatterProps
) {
  let k: keyof ScatterProps;
  for (k in oldProps) {
    if (oldProps[k] !== newProps[k] && REQUIRES_UPDATE[k]) {
      return true;
    }
  }
  return false;
}
```

The `[k in keyof ScatterProps]` tells the type checker that `REQUIRES_UPDATES` should have all the same properties as `ScatterProps`. If future you adds a new property to `ScatterProps`:

```
interface ScatterProps {
  // ...
  onDoubleClick: () => void;
}
```

Then this will produce an error in the definition of `REQUIRES_UPDATE`:

```
const REQUIRES_UPDATE: {[k in keyof ScatterProps]: boolean} = {
  //  ~~~~~~~~~~~~~~~ Property 'onDoubleClick' is missing in type
  // ...
};
```

This will certainly force the issue! Deleting or renaming a property will cause a similar error.

It's important that we used an object with boolean values here. Had we used an array:

```
const PROPS_REQUIRING_UPDATE: (keyof ScatterProps)[] = [
  'xs',
  'ys',
  // ...
];
```

then we would have been forced into the same fail open/fail closed choice.

Mapped types are ideal if you want one object to have exactly the same properties as another. As in this example, you can use this to make TypeScript enforce constraints on your code.

Things to Remember

- Use mapped types to keep related values and types synchronized.
- Consider using mapped types to force choices when adding new properties to an interface.

CHAPTER 3

Type Inference

For programming languages used in industry, “statically typed” and “explicitly typed” have traditionally been synonymous. C, C++, Java: they all made you write out your types. But academic languages never conflated these two things: languages like ML and Haskell have long had sophisticated type inference systems, and in the past decade this has begun to work its way into industry languages. C++ has added `auto`, and Java has added `var`.

TypeScript makes extensive use of type inference. Used well, this can dramatically reduce the number of type annotations your code requires to get full type safety. One of the easiest ways to tell a TypeScript beginner from a more experienced user is by the number of type annotations. An experienced TypeScript developer will use relatively few annotations (but use them to great effect), while a beginner may drown their code in redundant type annotations.

This chapter shows you some of the problems that can arise with type inference and how to fix them. After reading it, you should have a good understanding of how TypeScript infers types, when you still need to write type declarations, and when it’s a good idea to write type declarations even when a type can be inferred.

Item 19: Avoid Cluttering Your Code with Inferable Types

The first thing that many new TypeScript developers do when they convert a codebase from JavaScript is fill it with type annotations. TypeScript is about *types*, after all! But in TypeScript many annotations are unnecessary. Declaring types for all your variables is counterproductive and is considered poor style.

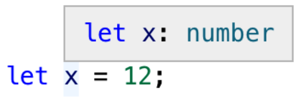
Don’t write:

```
let x: number = 12;
```

Instead, just write:

```
let x = 12;
```

If you mouse over x in your editor, you'll see that its type has been inferred as number (as shown in Figure 3-1).



```
let x: number
let x = 12;
```

Figure 3-1. A text editor showing that the inferred type of x is number.

The explicit type annotation is redundant. Writing it just adds noise. If you're unsure of the type, you can check it in your editor.

TypeScript will also infer the types of more complex objects. Instead of:

```
const person: {
  name: string;
  born: {
    where: string;
    when: string;
  };
  died: {
    where: string;
    when: string;
  }
} = {
  name: 'Sojourner Truth',
  born: {
    where: 'Swartekill, NY',
    when: 'c.1797',
  },
  died: {
    where: 'Battle Creek, MI',
    when: 'Nov. 26, 1883'
  }
};
```

you can just write:

```
const person = {
  name: 'Sojourner Truth',
  born: {
    where: 'Swartekill, NY',
    when: 'c.1797',
  },
  died: {
    where: 'Battle Creek, MI',
    when: 'Nov. 26, 1883'
  }
};
```

Again, the types are exactly the same. Writing the type in addition to the value just adds noise here. (Item 21 has more to say on the types inferred for object literals.)

What's true for objects is also true for arrays. TypeScript has no trouble figuring out the return type of this function based on its inputs and operations:

```
function square(nums: number[]) {
  return nums.map(x => x * x);
}
const squares = square([1, 2, 3, 4]); // Type is number[]
```

TypeScript may infer something more precise than what you expected. This is generally a good thing. For example:

```
const axis1: string = 'x';  // Type is string
const axis2 = 'y';  // Type is "y"
```

`"y"` is a more precise type for the `axis` variable. Item 21 gives an example of how this can fix a type error.

Allowing types to be inferred can also facilitate refactoring. Say you have a `Product` type and a function to log it:

```
interface Product {
  id: number;
  name: string;
  price: number;
}

function logProduct(product: Product) {
  const id: number = product.id;
  const name: string = product.name;
  const price: number = product.price;
  console.log(id, name, price);
}
```

At some point you learn that product IDs might have letters in them in addition to numbers. So you change the type of `id` in `Product`. Because you included explicit annotations on all the variables in `logProduct`, this produces an error:

```
interface Product {
  id: string;
  name: string;
  price: number;
}

function logProduct(product: Product) {
  const id: number = product.id;
     // ~~ Type 'string' is not assignable to type 'number'
  const name: string = product.name;
  const price: number = product.price;
  console.log(id, name, price);
}
```

Had you left off all the annotations in the `logProduct` function body, the code would have passed the type checker without modification.

A better implementation of `logProduct` would use destructuring assignment (Item 58):

```
function logProduct(product: Product) {
  const {id, name, price} = product;
  console.log(id, name, price);
}
```

This version allows the types of all the local variables to be inferred. The corresponding version with explicit type annotations is repetitive and cluttered:

```
function logProduct(product: Product) {
  const {id, name, price}: {id: string; name: string; price: number } = product;
  console.log(id, name, price);
}
```

Explicit type annotations are still required in some situations where TypeScript doesn't have enough context to determine a type on its own. You have seen one of these before: function parameters.

Some languages will infer types for parameters based on their eventual usage, but TypeScript does not. In TypeScript, a variable's type is generally determined when it is first introduced.

Ideal TypeScript code includes type annotations for function/method signatures but not for the local variables created in their bodies. This keeps noise to a minimum and lets readers focus on the implementation logic.

There are some situations where you can leave the type annotations off of function parameters, too. When there's a default value, for example:

```
function parseNumber(str: string, base=10) {
  // ...
}
```

Here the type of `base` is inferred as `number` because of the default value of `10`.

Parameter types can usually be inferred when the function is used as a callback for a library with type declarations. The declarations on `request` and `response` in this example using the express HTTP server library are not required:

```
// Don't do this:
app.get('/health', (request: express.Request, response: express.Response) => {
  response.send('OK');
});
```

```
// Do this:
app.get('/health', (request, response) => {
  response.send('OK');
});
```

Item 26 goes into more depth on how context is used in type inference.

There are a few situations where you may still want to specify a type even where it can be inferred.

One is when you define an object literal:

```
const elmo: Product = {
  name: 'Tickle Me Elmo',
  id: '048188 627152',
  price: 28.99,
};
```

When you specify a type on a definition like this, you enable excess property checking (Item 11). This can help catch errors, particularly for types with optional fields.

You also increase the odds that an error will be reported in the right place. If you leave off the annotation, a mistake in the object's definition will result in a type error where it's used, rather than where it's defined:

```
const furby = {
  name: 'Furby',
  id: 630509430963,
  price: 35,
};
logProduct(furby);
        // ~~~~~ Argument .. is not assignable to parameter of type 'Product'
        //         Types of property 'id' are incompatible
        //         Type 'number' is not assignable to type 'string'
```

With an annotation, you get a more concise error in the place where the mistake was made:

```
 const furby: Product = {
   name: 'Furby',
   id: 630509430963,
// ~~ Type 'number' is not assignable to type 'string'
   price: 35,
 };
 logProduct(furby);
```

Similar considerations apply to a function's return type. You may still want to annotate this even when it can be inferred to ensure that implementation errors don't leak out into uses of the function.

Say you have a function which retrieves a stock quote:

```
function getQuote(ticker: string) {
  return fetch(`https://quotes.example.com/?q=${ticker}`)
```

```
      .then(response => response.json());
}
```

You decide to add a cache to avoid duplicating network requests:

```
const cache: {[ticker: string]: number} = {};
function getQuote(ticker: string) {
  if (ticker in cache) {
    return cache[ticker];
  }
  return fetch(`https://quotes.example.com/?q=${ticker}`)
      .then(response => response.json())
      .then(quote => {
        cache[ticker] = quote;
        return quote;
      });
}
```

There's a mistake in this implementation: you should really be returning `Promise.resolve(cache[ticker])` so that `getQuote` always returns a Promise. The mistake will most likely produce an error…but in the code that calls `getQuote`, rather than in `getQuote` itself:

```
getQuote('MSFT').then(considerBuying);
              // ~~~~ Property 'then' does not exist on type
              //        'number | Promise<any>'
              //      Property 'then' does not exist on type 'number'
```

Had you annotated the intended return type (`Promise<number>`), the error would have been reported in the correct place:

```
const cache: {[ticker: string]: number} = {};
function getQuote(ticker: string): Promise<number> {
  if (ticker in cache) {
    return cache[ticker];
        // ~~~~~~~~~~~~~ Type 'number' is not assignable to 'Promise<number>'
  }
  // ...
}
```

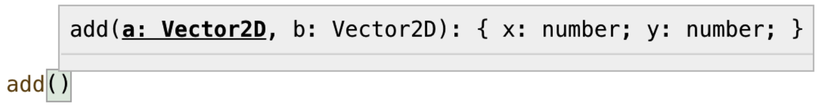
When you annotate the return type, it keeps implementation errors from manifesting as errors in user code. (See Item 25 for a discussion of async functions, which are an effective way to avoid this specific error with Promises.)

Writing out the return type may also help you think more clearly about your function: you should know what its input and output types are *before you implement it*. While the implementation may shift around a bit, the function's contract (its type signature) generally should not. This is similar in spirit to test-driven development (TDD), in which you write the tests that exercise a function before you implement it. Writing the full type signature first helps get you the function you want, rather than the one the implementation makes expedient.

A final reason to annotate return values is if you want to use a named type. You might choose not to write a return type for this function, for example:

```
interface Vector2D { x: number; y: number; }
function add(a: Vector2D, b: Vector2D) {
  return { x: a.x + b.x, y: a.y + b.y };
}
```

TypeScript infers the return type as `{ x: number; y: number; }`. This is compatible with `Vector2D`, but it may be surprising to users of your code when they see `Vector2D` as a type of the input and not of the output (as shown in Figure 3-2).



```
add(a: Vector2D, b: Vector2D): { x: number; y: number; }
add()
```

Figure 3-2. The parameters to the add function have named types, while the inferred return value does not.

If you annotate the return type, the presentation is more straightforward. And if you've written documentation on the type (Item 48) then it will be associated with the returned value as well. As the complexity of the inferred return type increases, it becomes increasingly helpful to provide a name.

If you are using a linter, the eslint rule `no-inferrable-types` (note the variant spelling) can help ensure that all your type annotations are really necessary.

Things to Remember

- Avoid writing type annotations when TypeScript can infer the same type.
- Ideally your code has type annotations in function/method signatures but not on local variables in their bodies.
- Consider using explicit annotations for object literals and function return types even when they can be inferred. This will help prevent implementation errors from surfacing in user code.

Item 20: Use Different Variables for Different Types

In JavaScript it's no problem to reuse a variable to hold a differently typed value for a different purpose:

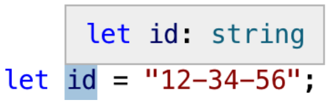
```
let id = "12-34-56";
fetchProduct(id);  // Expects a string
```

```
id = 123456;
fetchProductBySerialNumber(id);  // Expects a number
```

In TypeScript, this results in two errors:

```
    let id = "12-34-56";
    fetchProduct(id);

    id = 123456;
// ~~ '123456' is not assignable to type 'string'.
    fetchProductBySerialNumber(id);
                                 // ~~ Argument of type 'string' is not assignable to
                                 //    parameter of type 'number'
```

Hovering over the first `id` in your editor gives a hint as to what's going on (see Figure 3-3).



```
let id: string
let id = "12-34-56";
```

Figure 3-3. The inferred type of id is string.

Based on the value `"12-34-56"`, TypeScript has inferred `id`'s type as `string`. You can't assign a `number` to a `string` and hence the error.

This leads us to a key insight about variables in TypeScript: *while a variable's value can change, its type generally does not*. The one common way a type can change is to narrow (Item 22), but this involves a type getting smaller, not expanding to include new values. There are some important exceptions to this rule (Item 41), but they are the exceptions and not the rule.

How can you use this idea to fix the example? In order for `id`'s type to not change, it must be broad enough to encompass both `string`s and `number`s. This is the very definition of the union type, `string|number`:

```
let id: string|number = "12-34-56";
fetchProduct(id);

id = 123456;  // OK
fetchProductBySerialNumber(id);  // OK
```

This fixes the errors. It's interesting that TypeScript has been able to determine that `id` is really a `string` in the first call and really a `number` in the second. It has narrowed the union type based on the assignment.

While a union type does work, it may create more issues down the road. Union types are harder to work with than simple types like `string` or `number` because you usually have to check what they are before you do anything with them.

The better solution is to introduce a new variable:

```
const id = "12-34-56";
fetchProduct(id);

const serial = 123456;  // OK
fetchProductBySerialNumber(serial);  // OK
```

In the previous version, the first and second `id` were not semantically related to one another. They were only related by the fact that you reused a variable. This was confusing for the type checker and would be confusing for a human reader, too.

The version with two variables is better for a number of reasons:

- It disentangles two unrelated concepts (ID and serial number).
- It allows you to use more specific variable names.
- It improves type inference. No type annotations are needed.
- It results in simpler types (`string` and `number`, rather than `string|number`).
- It lets you declare the variables `const` rather than `let`. This makes them easier for people and the type checker to reason about.

Try to avoid type-changing variables. If you can use different names for different concepts, it will make your code clearer both to human readers and to the type checker.

This is not to be confused with "shadowed" variables as in this example:

```
const id = "12-34-56";
fetchProduct(id);

{
  const id = 123456;  // OK
  fetchProductBySerialNumber(id);  // OK
}
```

While these two `id`s share a name, they are actually two distinct variables with no relationship to one another. It's fine for them to have different types. While TypeScript is not confused by this, your human readers might be. In general it's better to use different names for different concepts. Many teams choose to disallow this sort of shadowing via linter rules.

This item focused on scalar values, but similar considerations apply to objects. For more on that, see Item 23.

Things to Remember

- While a variable's value can change, its type generally does not.

- To avoid confusion, both for human readers and for the type checker, avoid reusing variables for differently typed values.

Item 21: Understand Type Widening

As Item 7 explained, at runtime every variable has a single value. But at static analysis time, when TypeScript is checking your code, a variable has a set of *possible* values, namely, its type. When you initialize a variable with a constant but don't provide a type, the type checker needs to decide on one. In other words, it needs to decide on a set of possible values from the single value that you specified. In TypeScript, this process is known as *widening*. Understanding it will help you make sense of errors and make more effective use of type annotations.

Suppose you're writing a library to work with vectors. You write out a type for a 3D vector and a function to get the value of any of its components:

```
interface Vector3 { x: number; y: number; z: number; }
function getComponent(vector: Vector3, axis: 'x' | 'y' | 'z') {
  return vector[axis];
}
```

But when you try to use it, TypeScript flags an error:

```
let x = 'x';
let vec = {x: 10, y: 20, z: 30};
getComponent(vec, x);
               // ~ Argument of type 'string' is not assignable to
               //   parameter of type '"x" | "y" | "z"'
```

This code runs fine, so why the error?

The issue is that x's type is inferred as `string`, whereas the `getComponent` function expected a more specific type for its second argument. This is widening at work, and here it has led to an error.

This process is ambiguous in the sense that there are many possible types for any given value. In this statement, for example:

```
const mixed = ['x', 1];
```

what should the type of `mixed` be? Here are a few possibilities:

- `('x' | 1)[]`
- `['x', 1]`
- `[string, number]`
- `readonly [string, number]`
- `(string|number)[]`

- `readonly (string|number)[]`
- `[any, any]`
- `any[]`

Without more context, TypeScript has no way to know which one is "right." It has to guess at your intent. (In this case, it guesses `(string|number)[]`.) And smart as it is, TypeScript can't read your mind. It won't get this right 100% of the time. The result is inadvertent errors like the one we just looked at.

In the initial example, the type of `x` is inferred as `string` because TypeScript chooses to allow code like this:

```
let x = 'x';
x = 'a';
x = 'Four score and seven years ago...';
```

But it would also be valid JavaScript to write:

```
let x = 'x';
x = /x|y|z/;
x = ['x', 'y', 'z'];
```

In inferring the type of `x` as `string`, TypeScript attempts to strike a balance between specificity and flexibility. The general rule is that a variable's type shouldn't change after it's declared (Item 20), so `string` makes more sense than `string|RegExp` or `string|string[]` or `any`.

TypeScript gives you a few ways to control the process of widening. One is `const`. If you declare a variable with `const` instead of `let`, it gets a narrower type. In fact, using `const` fixes the error in our original example:

```
const x = 'x';  // type is "x"
let vec = {x: 10, y: 20, z: 30};
getComponent(vec, x);  // OK
```

Because `x` cannot be reassigned, TypeScript is able to infer a narrower type without risk of inadvertently flagging errors on subsequent assignments. And because the string literal type `"x"` is assignable to `"x"|"y"|"z"`, the code passes the type checker.

`const` isn't a panacea, however. For objects and arrays, there is still ambiguity. The `mixed` example here illustrates the issue for arrays: should TypeScript infer a tuple type? What type should it infer for the elements? Similar issues arise with objects. This code is fine in JavaScript:

```
const v = {
  x: 1,
};
v.x = 3;
v.x = '3';
```

```
v.y = 4;
v.name = 'Pythagoras';
```

The type of `v` could be inferred anywhere along the spectrum of specificity. At the specific end is `{readonly x: 1}`. More general is `{x: number}`. More general still would be `{[key: string]: number}` or `object`. In the case of objects, TypeScript's widening algorithm treats each element as though it were assigned with `let`. So the type of `v` comes out as `{x: number}`. This lets you reassign `v.x` to a different number, but not to a `string`. And it prevents you from adding other properties. (This is a good reason to build objects all at once, as explained in Item 23.)

So the last three statements are errors:

```
 const v = {
   x: 1,
 };
 v.x = 3;  // OK
 v.x = '3';
// ~ Type '"3"' is not assignable to type 'number'
 v.y = 4;
// ~ Property 'y' does not exist on type '{ x: number; }'
 v.name = 'Pythagoras';
// ~~~~ Property 'name' does not exist on type '{ x: number; }'
```

Again, TypeScript is trying to strike a balance between specificity and flexibility. It needs to infer a specific enough type to catch errors, but not so specific that it creates false positives. It does this by inferring a type of `number` for a property initialized to a value like `1`.

If you know better, there are a few ways to override TypeScript's default behavior. One is to supply an explicit type annotation:

```
const v: {x: 1|3|5} = {
  x: 1,
};  // Type is { x: 1 | 3 | 5; }
```

Another is to provide additional context to the type checker (e.g., by passing the value as the parameter of a function). For much more on the role of context in type inference, see Item 26.

A third way is with a `const` assertion. This is not to be confused with `let` and `const`, which introduce symbols in value space. This is a purely type-level construct. Look at the different inferred types for these variables:

```
const v1 = {
  x: 1,
  y: 2,
};  // Type is { x: number; y: number; }

const v2 = {
  x: 1 as const,
```

```
  y: 2,
};  // Type is { x: 1; y: number; }

const v3 = {
  x: 1,
  y: 2,
} as const;  // Type is { readonly x: 1; readonly y: 2; }
```

When you write `as const` after a value, TypeScript will infer the narrowest possible type for it. There is *no* widening. For true constants, this is typically what you want. You can also use `as const` with arrays to infer a tuple type:

```
const a1 = [1, 2, 3];  // Type is number[]
const a2 = [1, 2, 3] as const;  // Type is readonly [1, 2, 3]
```

If you're getting incorrect errors that you think are due to widening, consider adding some explicit type annotations or `const` assertions. Inspecting types in your editor is the key to building an intuition for this (see Item 6).

Things to Remember

- Understand how TypeScript infers a type from a constant by widening it.
- Familiarize yourself with the ways you can affect this behavior: `const`, type annotations, context, and `as const`.

Item 22: Understand Type Narrowing

The opposite of widening is narrowing. This is the process by which TypeScript goes from a broad type to a narrower one. Perhaps the most common example of this is null checking:

```
const el = document.getElementById('foo'); // Type is HTMLElement | null
if (el) {
  el // Type is HTMLElement
  el.innerHTML = 'Party Time'.blink();
} else {
  el // Type is null
  alert('No element #foo');
}
```

If `el` is `null`, then the code in the first branch won't execute. So TypeScript is able to exclude `null` from the type union within this block, resulting in a narrower type which is much easier to work with. The type checker is generally quite good at narrowing types in conditionals like these, though it can occasionally be thwarted by aliasing (Item 24).

You can also narrow a variable's type for the rest of a block by throwing or returning from a branch. For example:

```
const el = document.getElementById('foo'); // Type is HTMLElement | null
if (!el) throw new Error('Unable to find #foo');
el; // Now type is HTMLElement
el.innerHTML = 'Party Time'.blink();
```

There are many ways that you can narrow a type. Using `instanceof` works:

```
function contains(text: string, search: string|RegExp) {
  if (search instanceof RegExp) {
    search  // Type is RegExp
    return !!search.exec(text);
  }
  search  // Type is string
  return text.includes(search);
}
```

So does a property check:

```
interface A { a: number }
interface B { b: number }
function pickAB(ab: A | B) {
  if ('a' in ab) {
    ab // Type is A
  } else {
    ab // Type is B
  }
  ab // Type is A | B
}
```

Some built-in functions such as `Array.isArray` are able to narrow types:

```
function contains(text: string, terms: string|string[]) {
  const termList = Array.isArray(terms) ? terms : [terms];
  termList // Type is string[]
  // ...
}
```

TypeScript is generally quite good at tracking types through conditionals. Think twice before adding an assertion—it might be onto something that you're not! For example, this is the wrong way to exclude `null` from a union type:

```
const el = document.getElementById('foo'); // type is HTMLElement | null
if (typeof el === 'object') {
  el;  // Type is HTMLElement | null
}
```

Because `typeof null` is `"object"` in JavaScript, you have not, in fact, excluded `null` with this check! Similar surprises can come from falsy primitive values:

```
function foo(x?: number|string|null) {
  if (!x) {
    x;  // Type is string | number | null | undefined
  }
}
```

Because the empty string and `0` are both falsy, `x` could still be a `string` or `number` in that branch. TypeScript is right!

Another common way to help the type checker narrow your types is by putting an explicit "tag" on them:

```
interface UploadEvent { type: 'upload'; filename: string; contents: string }
interface DownloadEvent { type: 'download'; filename: string; }
type AppEvent = UploadEvent | DownloadEvent;

function handleEvent(e: AppEvent) {
  switch (e.type) {
    case 'download':
      e  // Type is DownloadEvent
      break;
    case 'upload':
      e;  // Type is UploadEvent
      break;
  }
}
```

This pattern is known as a "tagged union" or "discriminated union," and it is ubiquitous in TypeScript.

If TypeScript isn't able to figure out a type, you can even introduce a custom function to help it out:

```
function isInputElement(el: HTMLElement): el is HTMLInputElement {
  return 'value' in el;
}

function getElementContent(el: HTMLElement) {
  if (isInputElement(el)) {
    el; // Type is HTMLInputElement
    return el.value;
  }
  el; // Type is HTMLElement
  return el.textContent;
}
```

This is known as a "user-defined type guard." The `el is HTMLInputElement` as a return type tells the type checker that it can narrow the type of the parameter if the function returns true.

Some functions are able to use type guards to perform type narrowing across arrays or objects. If you do some lookups in an array, for instance, you may wind up with an array of nullable types:

```
const jackson5 = ['Jackie', 'Tito', 'Jermaine', 'Marlon', 'Michael'];
const members = ['Janet', 'Michael'].map(
  who => jackson5.find(n => n === who)
);  // Type is (string | undefined)[]
```

If you filter out the `undefined` values using `filter`, TypeScript isn't able to follow along:

```
const members = ['Janet', 'Michael'].map(
  who => jackson5.find(n => n === who)
).filter(who => who !== undefined);  // Type is (string | undefined)[]
```

But if you use a type guard, it can:

```
function isDefined<T>(x: T | undefined): x is T {
  return x !== undefined;
}
const members = ['Janet', 'Michael'].map(
  who => jackson5.find(n => n === who)
).filter(isDefined);  // Type is string[]
```

As always, inspecting types in your editor is key to building an intuition for how narrowing works.

Understanding how types in TypeScript narrow will help you build an intuition for how type inference works, make sense of errors, and generally have a more productive relationship with the type checker.

Things to Remember

- Understand how TypeScript narrows types based on conditionals and other types of control flow.
- Use tagged/discriminated unions and user-defined type guards to help the process of narrowing.

Item 23: Create Objects All at Once

As Item 20 explained, while a variable's value may change, its type in TypeScript generally does not. This makes some JavaScript patterns easier to model in TypeScript than others. In particular, it means that you should prefer creating objects all at once, rather than piece by piece.

Here's one way to create an object representing a two-dimensional point in JavaScript:

```
const pt = {};
pt.x = 3;
pt.y = 4;
```

In TypeScript, this will produce errors on each assignment:

```
const pt = {};
pt.x = 3;
// ~ Property 'x' does not exist on type '{}'
pt.y = 4;
// ~ Property 'y' does not exist on type '{}'
```

This is because the type of `pt` on the first line is inferred based on its value `{}`, and you may only assign to known properties.

You get the opposite problem if you define a `Point` interface:

```
interface Point { x: number; y: number; }
const pt: Point = {};
   // ~~ Type '{}' is missing the following properties from type 'Point': x, y
pt.x = 3;
pt.y = 4;
```

The solution is to define the object all at once:

```
const pt = {
  x: 3,
  y: 4,
};  // OK
```

If you must build the object piecemeal, you may use a type assertion (`as`) to silence the type checker:

```
const pt = {} as Point;
pt.x = 3;
pt.y = 4;  // OK
```

But the better way is by building the object all at once and using a declaration (see Item 9):

```
const pt: Point = {
  x: 3,
  y: 4,
};
```

If you need to build a larger object from smaller ones, avoid doing it in multiple steps:

```
const pt = {x: 3, y: 4};
const id = {name: 'Pythagoras'};
const namedPoint = {};
Object.assign(namedPoint, pt, id);
namedPoint.name;
        // ~~~~ Property 'name' does not exist on type '{}'
```

You can build the larger object all at once instead using the *object spread operator*, `...`:

```
const namedPoint = {...pt, ...id};
namedPoint.name;  // OK, type is string
```

You can also use the object spread operator to build up objects field by field in a type-safe way. The key is to use a new variable on every update so that each gets a new type:

```
const pt0 = {};
const pt1 = {...pt0, x: 3};
const pt: Point = {...pt1, y: 4};  // OK
```

While this is a roundabout way to build up such a simple object, it can be a useful technique for adding properties to an object and allowing TypeScript to infer a new type.

To conditionally add a property in a type-safe way, you can use object spread with `null` or `{}`, which add no properties:

```
declare let hasMiddle: boolean;
const firstLast = {first: 'Harry', last: 'Truman'};
const president = {...firstLast, ...(hasMiddle ? {middle: 'S'} : {})};
```

If you mouse over `president` in your editor, you'll see that its type is inferred as a union:

```
const president: {
    middle: string;
    first: string;
    last: string;
} | {
    first: string;
    last: string;
}
```

This may come as a surprise if you wanted `middle` to be an optional field. You can't read `middle` off this type, for example:

```
president.middle
       // ~~~~~~ Property 'middle' does not exist on type
       //        '{ first: string; last: string; }'
```

If you're conditionally adding multiple properties, the union does more accurately represent the set of possible values (Item 32). But an optional field would be easier to work with. You can get one with a helper:

```
function addOptional<T extends object, U extends object>(
  a: T, b: U | null
): T & Partial<U> {
  return {...a, ...b};
}

const president = addOptional(firstLast, hasMiddle ? {middle: 'S'} : null);
president.middle  // OK, type is string | undefined
```

Sometimes you want to build an object or array by transforming another one. In this case the equivalent of "building objects all at once" is using built-in functional constructs or utility libraries like Lodash rather than loops. See Item 27 for more on this.

Things to Remember

- Prefer to build objects all at once rather than piecemeal. Use object spread (`{...a, ...b}`) to add properties in a type-safe way.
- Know how to conditionally add properties to an object.

Item 24: Be Consistent in Your Use of Aliases

When you introduce a new name for a value:

```
const borough = {name: 'Brooklyn', location: [40.688, -73.979]};
const loc = borough.location;
```

you have created an *alias*. Changes to properties on the alias will be visible on the original value as well:

```
> loc[0] = 0;
> borough.location
[0, -73.979]
```

Aliases are the bane of compiler writers in all languages because they make control flow analysis difficult. If you're deliberate in your use of aliases, TypeScript will be able to understand your code better and help you find more real errors.

Suppose you have a data structure that represents a polygon:

```
interface Coordinate {
  x: number;
  y: number;
}

interface BoundingBox {
  x: [number, number];
  y: [number, number];
}

interface Polygon {
  exterior: Coordinate[];
  holes: Coordinate[][];
  bbox?: BoundingBox;
}
```

The geometry of the polygon is specified by the `exterior` and `holes` properties. The `bbox` property is an optimization that may or may not be present. You can use it to speed up a point-in-polygon check:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  if (polygon.bbox) {
    if (pt.x < polygon.bbox.x[0] || pt.x > polygon.bbox.x[1] ||
        pt.y < polygon.bbox.y[1] || pt.y > polygon.bbox.y[1]) {
      return false;
    }
  }

  // ... more complex check
}
```

This code works (and type checks) but is a bit repetitive: `polygon.bbox` appears five times in three lines! Here's an attempt to factor out an intermediate variable to reduce duplication:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const box = polygon.bbox;
  if (polygon.bbox) {
    if (pt.x < box.x[0] || pt.x > box.x[1] ||
        //     ~~~                ~~~  Object is possibly 'undefined'
        pt.y < box.y[1] || pt.y > box.y[1]) {
        //     ~~~                ~~~  Object is possibly 'undefined'
      return false;
    }
  }
  // ...
}
```

(I'm assuming you've enabled `strictNullChecks`.)

This code still works, so why the error? By factoring out the `box` variable, you've created an alias for `polygon.bbox`, and this has thwarted the control flow analysis that quietly worked in the first example.

You can inspect the types of `box` and `polygon.bbox` to see what's happening:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  polygon.bbox  // Type is BoundingBox | undefined
  const box = polygon.bbox;
  box  // Type is BoundingBox | undefined
  if (polygon.bbox) {
    polygon.bbox  // Type is BoundingBox
    box  // Type is BoundingBox | undefined
  }
}
```

The property check refines the type of `polygon.bbox` but not of `box` and hence the errors. This leads us to the golden rule of aliasing: *if you introduce an alias, use it consistently*.

Using `box` in the property check fixes the error:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const box = polygon.bbox;
  if (box) {
    if (pt.x < box.x[0] || pt.x > box.x[1] ||
        pt.y < box.y[1] || pt.y > box.y[1]) {  // OK
      return false;
    }
  }
  // ...
}
```

The type checker is happy now, but there's an issue for human readers. We're using two names for the same thing: box and bbox. This is a distinction without a difference (Item 36).

Object destructuring syntax rewards consistent naming with a more compact syntax. You can even use it on arrays and nested structures:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const {bbox} = polygon;
  if (bbox) {
    const {x, y} = bbox;
    if (pt.x < x[0] || pt.x > x[1] ||
        pt.y < x[0] || pt.y > y[1]) {
      return false;
    }
  }
  // ...
}
```

A few other points:

- This code would have required more property checks if the x and y properties had been optional, rather than the whole bbox property. We benefited from following the advice of Item 31, which discusses the importance of pushing null values to the perimeter of your types.
- An optional property was appropriate for bbox but would not have been appropriate for holes. If holes was optional, then it would be possible for it to be either missing or an empty array ([]). This would be a distinction without a difference. An empty array is a fine way to indicate "no holes."

In your interactions with the type checker, don't forget that aliasing can introduce confusion at runtime, too:

```
const {bbox} = polygon;
if (!bbox) {
  calculatePolygonBbox(polygon);  // Fills in polygon.bbox
  // Now polygon.bbox and bbox refer to different values!
}
```

TypeScript's control flow analysis tends to be quite good for local variables. But for properties you should be on guard:

```
function fn(p: Polygon) { /* ... */ }

polygon.bbox  // Type is BoundingBox | undefined
if (polygon.bbox) {
  polygon.bbox  // Type is BoundingBox
  fn(polygon);
  polygon.bbox  // Type is still BoundingBox
}
```

The call to `fn(polygon)` could very well un-set `polygon.bbox`, so it would be safer for the type to revert to `BoundingBox | undefined`. But this would get frustrating: you'd have to repeat your property checks every time you called a function. So TypeScript makes the pragmatic choice to assume the function does not invalidate its type refinements. But it *could*. If you'd factored out a local `bbox` variable instead of using `polygon.bbox`, the type of `bbox` would remain accurate, but it might no longer be the same value as `polygon.box`.

Things to Remember

- Aliasing can prevent TypeScript from narrowing types. If you create an alias for a variable, use it consistently.
- Use destructuring syntax to encourage consistent naming.
- Be aware of how function calls can invalidate type refinements on properties. Trust refinements on local variables more than on properties.

Item 25: Use async Functions Instead of Callbacks for Asynchronous Code

Classic JavaScript modeled asynchronous behavior using callbacks. This leads to the infamous "pyramid of doom":

```
fetchURL(url1, function(response1) {
  fetchURL(url2, function(response2) {
    fetchURL(url3, function(response3) {
      // ...
      console.log(1);
    });
    console.log(2);
  });
  console.log(3);
});
console.log(4);
```

```
// Logs:
// 4
// 3
// 2
// 1
```

As you can see from the logs, the execution order is the opposite of the code order. This makes callback code hard to read. It gets even more confusing if you want to run the requests in parallel or bail when an error occurs.

ES2015 introduced the concept of a Promise to break the pyramid of doom. A Promise represents something that will be available in the future (they're also sometimes called "futures"). Here's the same code using Promises:

```
const page1Promise = fetch(url1);
page1Promise.then(response1 => {
  return fetch(url2);
}).then(response2 => {
  return fetch(url3);
}).then(response3 => {
  // ...
}).catch(error => {
  // ...
});
```

Now there's less nesting, and the execution order more directly matches the code order. It's also easier to consolidate error handling and use higher-order tools like `Promise.all`.

ES2017 introduced the `async` and `await` keywords to make things even simpler:

```
async function fetchPages() {
  const response1 = await fetch(url1);
  const response2 = await fetch(url2);
  const response3 = await fetch(url3);
  // ...
}
```

The `await` keyword pauses execution of the `fetchPages` function until each Promise resolves. Within an `async` function, `await`ing a Promise that throws an exception. This lets you use the usual try/catch machinery:

```
async function fetchPages() {
  try {
    const response1 = await fetch(url1);
    const response2 = await fetch(url2);
    const response3 = await fetch(url3);
    // ...
  } catch (e) {
    // ...
  }
}
```

When you target ES5 or earlier, the TypeScript compiler will perform some elaborate transformations to make `async` and `await` work. In other words, whatever your runtime, with TypeScript you can use `async`/`await`.

There are a few good reasons to prefer Promises or `async`/`await` to callbacks:

- Promises are easier to compose than callbacks.
- Types are able to flow through Promises more easily than callbacks.

If you want to fetch the pages in parallel, for example, you can compose Promises with `Promise.all`:

```
async function fetchPages() {
  const [response1, response2, response3] = await Promise.all([
    fetch(url1), fetch(url2), fetch(url3)
  ]);
  // ...
}
```

Using destructuring assignment with `await` is particularly nice in this context.

TypeScript is able to infer the types of each of the three `response` variables as `Response`. The equivalent code to do the requests in parallel with callbacks requires more machinery and a type annotation:

```
function fetchPagesCB() {
  let numDone = 0;
  const responses: string[] = [];
  const done = () => {
    const [response1, response2, response3] = responses;
    // ...
  };
  const urls = [url1, url2, url3];
  urls.forEach((url, i) => {
    fetchURL(url, r => {
      responses[i] = url;
      numDone++;
      if (numDone === urls.length) done();
    });
  });
}
```

Extending this to include error handling or to be as generic as `Promise.all` is challenging.

Type inference also works well with `Promise.race`, which resolves when the first of its input Promises resolves. You can use this to add timeouts to Promises in a general way:

```
function timeout(millis: number): Promise<never> {
  return new Promise((resolve, reject) => {
     setTimeout(() => reject('timeout'), millis);
  });
}

async function fetchWithTimeout(url: string, ms: number) {
  return Promise.race([fetch(url), timeout(ms)]);
}
```

The return type of `fetchWithTimeout` is inferred as `Promise<Response>`, no type annotations required. It's interesting to dig into why this works: the return type of `Promise.race` is the union of the types of its inputs, in this case `Promise<Response | never>`. But taking a union with `never` (the empty set) is a no-op, so this gets simplified to `Promise<Response>`. When you work with Promises, all of TypeScript's type inference machinery works to get you the right types.

There are some times when you need to use raw Promises, notably when you are wrapping a callback API like `setTimeout`. But if you have a choice, you should generally prefer `async`/`await` to raw Promises for two reasons:

- It typically produces more concise and straightforward code.
- It enforces that `async` functions always return Promises.

An `async` function always returns a `Promise`, even if it doesn't involve `await`ing anything. TypeScript can help you build an intuition for this:

```
// function getNumber(): Promise<number>
async function getNumber() {
  return 42;
}
```

You can also create `async` arrow functions:

```
const getNumber = async () => 42;  // Type is () => Promise<number>
```

The raw Promise equivalent is:

```
const getNumber = () => Promise.resolve(42);  // Type is () => Promise<number>
```

While it may seem odd to return a Promise for an immediately available value, this actually helps enforce an important rule: a function should either always be run synchronously or always be run asynchronously. It should never mix the two. For example, what if you want to add a cache to the `fetchURL` function? Here's an attempt:

```
// Don't do this!
const _cache: {[url: string]: string} = {};
function fetchWithCache(url: string, callback: (text: string) => void) {
  if (url in _cache) {
    callback(_cache[url]);
  } else {
    fetchURL(url, text => {
      _cache[url] = text;
```

```
        callback(text);
      });
    }
  }
```

While this may seem like an optimization, the function is now extremely difficult for a client to use:

```
let requestStatus: 'loading' | 'success' | 'error';
function getUser(userId: string) {
  fetchWithCache(`/user/${userId}`, profile => {
    requestStatus = 'success';
  });
  requestStatus = 'loading';
}
```

What will the value of `requestStatus` be after calling `getUser`? It depends entirely on whether the profile is cached. If it's not, `requestStatus` will be set to "success." If it is, it'll get set to "success" and then set back to "loading." Oops!

Using `async` for both functions enforces consistent behavior:

```
const _cache: {[url: string]: string} = {};
async function fetchWithCache(url: string) {
  if (url in _cache) {
    return _cache[url];
  }
  const response = await fetch(url);
  const text = await response.text();
  _cache[url] = text;
  return text;
}

let requestStatus: 'loading' | 'success' | 'error';
async function getUser(userId: string) {
  requestStatus = 'loading';
  const profile = await fetchWithCache(`/user/${userId}`);
  requestStatus = 'success';
}
```

Now it's completely transparent that `requestStatus` will end in "success." It's easy to accidentally produce half-synchronous code with callbacks or raw Promises, but difficult with `async`.

Note that if you return a Promise from an `async` function, it will not get wrapped in another Promise: the return type will be `Promise<T>` rather than `Promise<Promise<T>>`. Again, TypeScript will help you build an intuition for this:

```
// Function getJSON(url: string): Promise<any>
async function getJSON(url: string) {
  const response = await fetch(url);
  const jsonPromise = response.json();  // Type is Promise<any>
  return jsonPromise;
}
```

Things to Remember

- Prefer Promises to callbacks for better composability and type flow.
- Prefer `async` and `await` to raw Promises when possible. They produce more concise, straightforward code and eliminate whole classes of errors.
- If a function returns a Promise, declare it `async`.

Item 26: Understand How Context Is Used in Type Inference

TypeScript doesn't just infer types based on values. It also considers the context in which the value occurs. This usually works well but can sometimes lead to surprises. Understanding how context is used in type inference will help you identify and work around these surprises when they do occur.

In JavaScript you can factor an expression out into a constant without changing the behavior of your code (so long as you don't alter execution order). In other words, these two statements are equivalent:

```
// Inline form
setLanguage('JavaScript');

// Reference form
let language = 'JavaScript';
setLanguage(language);
```

In TypeScript, this refactor still works:

```
function setLanguage(language: string) { /* ... */ }

setLanguage('JavaScript');  // OK

let language = 'JavaScript';
setLanguage(language);  // OK
```

Now suppose you take to heart the advice of Item 33 and replace the string type with a more precise union of string literal types:

```
type Language = 'JavaScript' | 'TypeScript' | 'Python';
function setLanguage(language: Language) { /* ... */ }

setLanguage('JavaScript');  // OK

let language = 'JavaScript';
setLanguage(language);
```

```
            // ~~~~~~~~ Argument of type 'string' is not assignable
            //          to parameter of type 'Language'
```

What went wrong? With the inline form, TypeScript knows from the function declaration that the parameter is supposed to be of type `Language`. The string literal `'JavaScript'` is assignable to this type, so this is OK. But when you factor out a variable, TypeScript must infer its type at the time of assignment. In this case it infers `string`, which is not assignable to `Language`. Hence the error.

(Some languages are able to infer types for variables based on their eventual usage. But this can also be confusing. Anders Hejlsberg, the creator of TypeScript, refers to it as "spooky action at a distance." By and large, TypeScript determines the type of a variable when it is first introduced. For a notable exception to this rule, see Item 41.)

There are two good ways to solve this problem. One is to constrain the possible values of `language` with a type declaration:

```
let language: Language = 'JavaScript';
setLanguage(language);  // OK
```

This also has the benefit of flagging an error if there's a typo in the language—for example `'Typescript'` (it should be a capital "S").

The other solution is to make the variable constant:

```
const language = 'JavaScript';
setLanguage(language);  // OK
```

By using `const`, we've told the type checker that this variable cannot change. So TypeScript can infer a more precise type for `language`, the string literal type `"JavaScript"`. This is assignable to `Language` so the code type checks. Of course, if you do need to reassign `language`, then you'll need to use the type declaration. (For more on this, see Item 21.)

The fundamental issue here is that we've separated the value from the context in which it's used. Sometimes this is OK, but often it is not. The rest of this item walks through a few cases where this loss of context can cause errors and shows you how to fix them.

Tuple Types

In addition to string literal types, problems can come up with tuple types. Suppose you're working with a map visualization that lets you programmatically pan the map:

```
// Parameter is a (latitude, longitude) pair.
function panTo(where: [number, number]) { /* ... */ }

panTo([10, 20]);  // OK

const loc = [10, 20];
```

```
panTo(loc);
//    ~~~ Argument of type 'number[]' is not assignable to
//        parameter of type '[number, number]'
```

As before, you've separated a value from its context. In the first instance `[10, 20]` is assignable to the tuple type `[number, number]`. In the second, TypeScript infers the type of `loc` as `number[]` (i.e., an array of numbers of unknown length). This is not assignable to the tuple type, since many arrays have the wrong number of elements.

So how can you fix this error without resorting to `any`? You've already declared it `const`, so that won't help. But you can still provide a type declaration to let TypeScript know precisely what you mean:

```
const loc: [number, number] = [10, 20];
panTo(loc);  // OK
```

Another way is to provide a "const context." This tells TypeScript that you intend the value to be deeply constant, rather than the shallow constant that `const` gives:

```
const loc = [10, 20] as const;
panTo(loc);
   // ~~~ Type 'readonly [10, 20]' is 'readonly'
   //     and cannot be assigned to the mutable type '[number, number]'
```

If you hover over `loc` in your editor, you'll see that its type is now inferred as `readonly [10, 20]`, rather than `number[]`. Unfortunately this is *too* precise! The type signature of `panTo` makes no promises that it won't modify the contents of its `where` parameter. Since the `loc` parameter has a `readonly` type, this won't do. The best solution here is to add a `readonly` annotation to the `panTo` function:

```
function panTo(where: readonly [number, number]) { /* ... */ }
const loc = [10, 20] as const;
panTo(loc);  // OK
```

If the type signature is outside your control, then you'll need to use an annotation.

`const` contexts can neatly solve issues around losing context in inference, but they do have an unfortunate downside: if you make a mistake in the definition (say you add a third element to the tuple) then the error will be flagged at the call site, not at the definition. This may be confusing, especially if the error occurs in a deeply nested object:

```
const loc = [10, 20, 30] as const;  // error is really here.
panTo(loc);
//    ~~~ Argument of type 'readonly [10, 20, 30]' is not assignable to
//        parameter of type 'readonly [number, number]'
//          Types of property 'length' are incompatible
//            Type '3' is not assignable to type '2'
```

Objects

The problem of separating a value from its context also comes up when you factor out a constant from a larger object that contains some string literals or tuples. For example:

```
type Language = 'JavaScript' | 'TypeScript' | 'Python';
interface GovernedLanguage {
  language: Language;
  organization: string;
}

function complain(language: GovernedLanguage) { /* ... */ }

complain({ language: 'TypeScript', organization: 'Microsoft' });  // OK

const ts = {
  language: 'TypeScript',
  organization: 'Microsoft',
};
complain(ts);
//       ~~ Argument of type '{ language: string; organization: string; }'
//            is not assignable to parameter of type 'GovernedLanguage'
//          Types of property 'language' are incompatible
//            Type 'string' is not assignable to type 'Language'
```

In the `ts` object, the type of `language` is inferred as `string`. As before, the solution is to add a type declaration (`const ts: GovernedLanguage = ...`) or use a const assertion (`as const`).

Callbacks

When you pass a callback to another function, TypeScript uses context to infer the parameter types of the callback:

```
function callWithRandomNumbers(fn: (n1: number, n2: number) => void) {
  fn(Math.random(), Math.random());
}

callWithRandomNumbers((a, b) => {
  a;  // Type is number
  b;  // Type is number
  console.log(a + b);
});
```

The types of `a` and `b` are inferred as `number` because of the type declaration for `callWithRandom`. If you factor the callback out into a constant, you lose that context and get `noImplicitAny` errors:

```
const fn = (a, b) => {
         // ~    Parameter 'a' implicitly has an 'any' type
```

```
        //    ~ Parameter 'b' implicitly has an 'any' type
  console.log(a + b);
}
callWithRandomNumbers(fn);
```

The solution is either to add type annotations to the parameters:

```
const fn = (a: number, b: number) => {
  console.log(a + b);
}
callWithRandomNumbers(fn);
```

or to apply a type declaration to the entire function expression if one is available. See Item 12.

Things to Remember

- Be aware of how context is used in type inference.
- If factoring out a variable introduces a type error, consider adding a type declaration.
- If the variable is truly a constant, use a const assertion (`as const`). But be aware that this may result in errors surfacing at use, rather than definition.

Item 27: Use Functional Constructs and Libraries to Help Types Flow

JavaScript has never included the sort of standard library you find in Python, C, or Java. Over the years many libraries have tried to fill the gap. jQuery provided helpers not just for interacting with the DOM but also for iterating and mapping over objects and arrays. Underscore focused more on providing general utility functions, and Lodash built on this effort. Today libraries like Ramda continue to bring ideas from functional programming into the JavaScript world.

Some features from these libraries, such as `map`, `flatMap`, `filter`, and `reduce`, have made it into the JavaScript language itself. While these constructs (and the other ones provided by Lodash) are helpful in JavaScript and often preferable to a hand-rolled loop, this advantage tends to get even more lopsided when you add TypeScript to the mix. This is because their type declarations ensure that types flow through these constructs. With hand-rolled loops, you're responsible for the types yourself.

For example, consider parsing some CSV data. You could do it in plain JavaScript in a somewhat imperative style:

```
const csvData = "...";
const rawRows = csvData.split('\n');
const headers = rawRows[0].split(',');
const rows = rawRows.slice(1).map(rowStr => {
  const row = {};
  rowStr.split(',').forEach((val, j) => {
    row[headers[j]] = val;
  });
  return row;
});
```

More functionally minded JavaScripters might prefer to build the row objects with `reduce`:

```
const rows = rawRows.slice(1)
    .map(rowStr => rowStr.split(',').reduce(
        (row, val, i) => (row[headers[i]] = val, row),
        {}));
```

This version saves three lines (almost 20 non-whitespace characters!) but may be more cryptic depending on your sensibilities. Lodash's `zipObject` function, which forms an object by "zipping" up a keys and values array, can tighten it even further:

```
import _ from 'lodash';
const rows = rawRows.slice(1)
    .map(rowStr => _.zipObject(headers, rowStr.split(',')));
```

I find this the clearest of all. But is it worth the cost of adding a dependency on a third-party library to your project? If you're not using a bundler and the overhead of doing this is significant, then the answer may be "no."

When you add TypeScript to the mix, it starts to tip the balance more strongly in favor of the Lodash solution.

Both vanilla JS versions of the CSV parser produce the same error in TypeScript:

```
const rowsA = rawRows.slice(1).map(rowStr => {
  const row = {};
  rowStr.split(',').forEach((val, j) => {
    row[headers[j]] = val;
 // ~~~~~~~~~~~~~~~ No index signature with a parameter of
 //                 type 'string' was found on type '{}'
  });
  return row;
});
const rowsB = rawRows.slice(1)
  .map(rowStr => rowStr.split(',').reduce(
      (row, val, i) => (row[headers[i]] = val, row),
                     // ~~~~~~~~~~~~~~~ No index signature with a parameter of
                     //                 type 'string' was found on type '{}'
      {}));
```

The solution in each case is to provide a type annotation for `{}`, either `{[column: string]: string}` or `Record<string, string>`.

The Lodash version, on the other hand, passes the type checker without modification:

```
const rows = rawRows.slice(1)
    .map(rowStr => _.zipObject(headers, rowStr.split(',')));
    // Type is _.Dictionary<string>[]
```

Dictionary is a Lodash type alias. Dictionary<string> is the same as {[key: string]: string} or Record<string, string>. The important thing here is that the type of rows is exactly correct, no type annotations needed.

These advantages get more pronounced as your data munging gets more elaborate. For example, suppose you have a list of the rosters for all the NBA teams:

```
interface BasketballPlayer {
  name: string;
  team: string;
  salary: number;
}
declare const rosters: {[team: string]: BasketballPlayer[]};
```

To build a flat list using a loop, you might use concat with an array. This code runs fine but does not type check:

```
let allPlayers = [];
 // ~~~~~~~~~~ Variable 'allPlayers' implicitly has type 'any[]'
 //            in some locations where its type cannot be determined
for (const players of Object.values(rosters)) {
  allPlayers = allPlayers.concat(players);
           // ~~~~~~~~~~ Variable 'allPlayers' implicitly has an 'any[]' type
}
```

To fix the error you need to add a type annotation to allPlayers:

```
let allPlayers: BasketballPlayer[] = [];
for (const players of Object.values(rosters)) {
  allPlayers = allPlayers.concat(players);  // OK
}
```

But a better solution is to use Array.prototype.flat:

```
const allPlayers = Object.values(rosters).flat();
// OK, type is BasketballPlayer[]
```

The flat method flattens a multidimensional array. Its type signature is something like T[][] => T[]. This version is the most concise and requires no type annotations. As an added bonus you can use const instead of let to prevent future mutations to the allPlayers variable.

Say you want to start with allPlayers and make a list of the highest-paid players on each team ordered by salary.

Here's a solution without Lodash. It requires a type annotation where you don't use functional constructs:

```
const teamToPlayers: {[team: string]: BasketballPlayer[]} = {};
for (const player of allPlayers) {
  const {team} = player;
  teamToPlayers[team] = teamToPlayers[team] || [];
  teamToPlayers[team].push(player);
}

for (const players of Object.values(teamToPlayers)) {
  players.sort((a, b) => b.salary - a.salary);
}

const bestPaid = Object.values(teamToPlayers).map(players => players[0]);
bestPaid.sort((playerA, playerB) => playerB.salary - playerA.salary);
console.log(bestPaid);
```

Here's the output:

```
[
  { team: 'GSW', salary: 37457154, name: 'Stephen Curry' },
  { team: 'HOU', salary: 35654150, name: 'Chris Paul' },
  { team: 'LAL', salary: 35654150, name: 'LeBron James' },
  { team: 'OKC', salary: 35654150, name: 'Russell Westbrook' },
  { team: 'DET', salary: 32088932, name: 'Blake Griffin' },
  ...
]
```

Here's the equivalent with Lodash:

```
const bestPaid = _(allPlayers)
  .groupBy(player => player.team)
  .mapValues(players => _.maxBy(players, p => p.salary)!)
  .values()
  .sortBy(p => -p.salary)
  .value()  // Type is BasketballPlayer[]
```

In addition to being half the length, this code is clearer and requires only a single non-null assertion (the type checker doesn't know that the `players` array passed to `_.maxBy` is non-empty). It makes use of a "chain," a concept in Lodash and Underscore that lets you write a sequence of operations in a more natural order. Instead of writing:

```
_.a(_.b(_.c(v)))
```

you write:

```
_(v).a().b().c().value()
```

The `_(v)` "wraps" the value, and the `.value()` "unwraps" it.

You can inspect each function call in the chain to see the type of the wrapped value. It's always correct.

Even some of the quirkier shorthands in Lodash can be modeled accurately in TypeScript. For instance, why would you want to use `_.map` instead of the built-in

`Array.prototype.map`? One reason is that instead of passing in a callback you can pass in the name of a property. These calls all produce the same result:

```
const namesA = allPlayers.map(player => player.name)  // Type is string[]
const namesB = _.map(allPlayers, player => player.name)  // Type is string[]
const namesC = _.map(allPlayers, 'name');  // Type is string[]
```

It's a testament to the sophistication of TypeScript's type system that it can model a construct like this accurately, but it naturally falls out of the combination of string literal types and index types (see Item 14). If you're used to C++ or Java, this sort of type inference can feel quite magical!

```
const salaries = _.map(allPlayers, 'salary');  // Type is number[]
const teams = _.map(allPlayers, 'team');  // Type is string[]
const mix = _.map(allPlayers, Math.random() < 0.5 ? 'name' : 'salary');
  // Type is (string | number)[]
```

It's not a coincidence that types flow so well through built-in functional constructs and those in libraries like Lodash. By avoiding mutation and returning new values from every call, they are able to produce new types as well (Item 20). And to a large extent, the development of TypeScript has been driven by an attempt to accurately model the behavior of JavaScript libraries in the wild. Take advantage of all this work and use them!

Things to Remember

- Use built-in functional constructs and those in utility libraries like Lodash instead of hand-rolled constructs to improve type flow, increase legibility, and reduce the need for explicit type annotations.

CHAPTER 4

Type Design

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

—Fred Brooks, *The Mythical Man Month*

The language in Fred Brooks's quote is dated, but the sentiment remains true: code is difficult to understand if you can't see the data or data types on which it operates. This is one of the great advantages of a type system: by writing out types, you make them visible to readers of your code. And this makes your code understandable.

Other chapters cover the nuts and bolts of TypeScript types: using them, inferring them, and writing declarations with them. This chapter discusses the design of the types themselves. The examples in this chapter are all written with TypeScript in mind, but most of the ideas are more broadly applicable.

If you write your types well, then with any luck your flowcharts will be obvious, too.

Item 28: Prefer Types That Always Represent Valid States

If you design your types well, your code should be straightforward to write. But if you design your types poorly, no amount of cleverness or documentation will save you. Your code will be confusing and bug prone.

A key to effective type design is crafting types that can only represent a valid state. This item walks through a few examples of how this can go wrong and shows you how to fix them.

Suppose you're building a web application that lets you select a page, loads the content of that page, and then displays it. You might write the state like this:

```
interface State {
  pageText: string;
  isLoading: boolean;
  error?: string;
}
```

When you write your code to render the page, you need to consider all of these fields:

```
function renderPage(state: State) {
  if (state.error) {
    return `Error! Unable to load ${currentPage}: ${state.error}`;
  } else if (state.isLoading) {
    return `Loading ${currentPage}...`;
  }
  return `<h1>${currentPage}</h1>\n${state.pageText}`;
}
```

Is this right, though? What if `isLoading` and `error` are both set? What would that mean? Is it better to display the loading message or the error message? It's hard to say! There's not enough information available.

Or what if you're writing a `changePage` function? Here's an attempt:

```
async function changePage(state: State, newPage: string) {
  state.isLoading = true;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`Unable to load ${newPage}: ${response.statusText}`);
    }
    const text = await response.text();
    state.isLoading = false;
    state.pageText = text;
  } catch (e) {
    state.error = '' + e;
  }
}
```

There are many problems with this! Here are a few:

- We forgot to set `state.isLoading` to `false` in the error case.
- We didn't clear out `state.error`, so if the previous request failed, then you'll keep seeing that error message instead of a loading message.
- If the user changes pages again while the page is loading, who knows what will happen. They might see a new page and then an error, or the first page and not the second depending on the order in which the responses come back.

The problem is that the state includes both too little information (which request failed? which is loading?) and too much: the `State` type allows both `isLoading` and `error` to be set, even though this represents an invalid state. This makes both `render()` and `changePage()` impossible to implement well.

Here's a better way to represent the application state:

```
interface RequestPending {
  state: 'pending';
}
interface RequestError {
  state: 'error';
  error: string;
}
interface RequestSuccess {
  state: 'ok';
  pageText: string;
}
type RequestState = RequestPending | RequestError | RequestSuccess;

interface State {
  currentPage: string;
  requests: {[page: string]: RequestState};
}
```

This uses a tagged union (also known as a "discriminated union") to explicitly model the different states that a network request can be in. This version of the state is three to four times longer, but it has the enormous advantage of not admitting invalid states. The current page is modeled explicitly, as is the state of every request that you issue. As a result, the `renderPage` and `changePage` functions are easy to implement:

```
function renderPage(state: State) {
  const {currentPage} = state;
  const requestState = state.requests[currentPage];
  switch (requestState.state) {
    case 'pending':
      return `Loading ${currentPage}...`;
    case 'error':
      return `Error! Unable to load ${currentPage}: ${requestState.error}`;
    case 'ok':
      return `<h1>${currentPage}</h1>\n${requestState.pageText}`;
  }
}

async function changePage(state: State, newPage: string) {
  state.requests[newPage] = {state: 'pending'};
  state.currentPage = newPage;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`Unable to load ${newPage}: ${response.statusText}`);
    }
    const pageText = await response.text();
    state.requests[newPage] = {state: 'ok', pageText};
  } catch (e) {
    state.requests[newPage] = {state: 'error', error: '' + e};
    }
  }
```

The ambiguity from the first implementation is entirely gone: it's clear what the current page is, and every request is in exactly one state. If the user changes the page after a request has been issued, that's no problem either. The old request still completes, but it doesn't affect the UI.

For a simpler but more dire example, consider the fate of Air France Flight 447, an Airbus 330 that disappeared over the Atlantic on June 1, 2009. The Airbus was a fly-by-wire aircraft, meaning that the pilots' control inputs went through a computer system before affecting the physical control surfaces of the plane. In the wake of the crash there were many questions raised about the wisdom of relying on computers to make such life-and-death decisions. Two years later when the black box recorders were recovered, they revealed many factors that led to the crash. But a key one was bad state design.

The cockpit of the Airbus 330 had a separate set of controls for the pilot and copilot. The "side sticks" controlled the angle of attack. Pulling back would send the airplane into a climb, while pushing forward would make it dive. The Airbus 330 used a system called "dual input" mode, which let the two side sticks move independently. Here's how you might model its state in TypeScript:

```
interface CockpitControls {
  /** Angle of the left side stick in degrees, 0 = neutral, + = forward */
  leftSideStick: number;
  /** Angle of the right side stick in degrees, 0 = neutral, + = forward */
  rightSideStick: number;
}
```

Suppose you were given this data structure and asked to write a `getStickSetting` function that computed the current stick setting. How would you do it?

One way would be to assume that the pilot (who sits on the left) is in control:

```
function getStickSetting(controls: CockpitControls) {
  return controls.leftSideStick;
}
```

But what if the copilot has taken control? Maybe you should use whichever stick is away from zero:

```
function getStickSetting(controls: CockpitControls) {
  const {leftSideStick, rightSideStick} = controls;
  if (leftSideStick === 0) {
    return rightSideStick;
  }
  return leftSideStick;
}
```

But there's a problem with this implementation: we can only be confident returning the left setting if the right one is neutral. So you should check for that:

```
function getStickSetting(controls: CockpitControls) {
  const {leftSideStick, rightSideStick} = controls;
  if (leftSideStick === 0) {
    return rightSideStick;
  } else if (rightSideStick === 0) {
    return leftSideStick;
  }
  // ???
}
```

What do you do if they're both non-zero? Hopefully they're about the same, in which case you could just average them:

```
function getStickSetting(controls: CockpitControls) {
  const {leftSideStick, rightSideStick} = controls;
  if (leftSideStick === 0) {
    return rightSideStick;
  } else if (rightSideStick === 0) {
    return leftSideStick;
  }
  if (Math.abs(leftSideStick - rightSideStick) < 5) {
    return (leftSideStick + rightSideStick) / 2;
  }
  // ???
}
```

But what if they're not? Can you throw an error? Not really: the ailerons need to be set at some angle!

On Air France 447, the copilot silently pulled back on his side stick as the plane entered a storm. It gained altitude but eventually lost speed and entered a stall, a condition in which the plane is moving too slowly to effectively generate lift. It began to drop.

To escape a stall, pilots are trained to push the controls forward to make the plane dive and regain speed. This is exactly what the pilot did. But the copilot was still silently pulling back on his side stick. And the Airbus function looked like this:

```
function getStickSetting(controls: CockpitControls) {
  return (controls.leftSideStick + controls.rightSideStick) / 2;
}
```

Even though the pilot pushed the stick fully forward, it averaged out to nothing. He had no idea why the plane wasn't diving. By the time the copilot revealed what he'd done, the plane had lost too much altitude to recover and it crashed into the ocean, killing all 228 people on board.

The point of all this is that there is no good way to implement `getStickSetting` given that input! The function has been set up to fail. In most planes the two sets of controls are mechanically connected. If the copilot pulls back, the pilot's controls will also pull back. The state of these controls is simple to express:

```
interface CockpitControls {
  /** Angle of the stick in degrees, 0 = neutral, + = forward */
  stickAngle: number;
}
```

And now, as in the Fred Brooks quote from the start of the chapter, our flowcharts are obvious. You don't need a `getStickSetting` function at all.

As you design your types, take care to think about which values you are including and which you are excluding. If you only allow values that represent valid states, your code will be easier to write and TypeScript will have an easier time checking it. This is a very general principle, and several of the other items in this chapter will cover specific manifestations of it.

Things to Remember

- Types that represent both valid and invalid states are likely to lead to confusing and error-prone code.
- Prefer types that only represent valid states. Even if they are longer or harder to express, they will save you time and pain in the end!

Item 29: Be Liberal in What You Accept and Strict in What You Produce

This idea is known as the *robustness principle* or *Postel's Law*, after Jon Postel, who wrote it in the context of TCP:

> TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.

A similar rule applies to the contracts for functions. It's fine for your functions to be broad in what they accept as inputs, but they should generally be more specific in what they produce as outputs.

As an example, a 3D mapping API might provide a way to position the camera and to calculate a viewport for a bounding box:

```
declare function setCamera(camera: CameraOptions): void;
declare function viewportForBounds(bounds: LngLatBounds): CameraOptions;
```

It is convenient that the result of `viewportForBounds` can be passed directly to `setCamera` to position the camera.

Let's look at the definitions of these types:

```
interface CameraOptions {
  center?: LngLat;
  zoom?: number;
  bearing?: number;
  pitch?: number;
}
type LngLat =
  { lng: number; lat: number; } |
  { lon: number; lat: number; } |
  [number, number];
```

The fields in `CameraOptions` are all optional because you might want to set just the center or zoom without changing the bearing or pitch. The `LngLat` type also makes `setCamera` liberal in what it accepts: you can pass in a `{lng, lat}` object, a `{lon, lat}` object, or a `[lng, lat]` pair if you're confident you got the order right. These accommodations make the function easy to call.

The `viewportForBounds` function takes in another "liberal" type:

```
type LngLatBounds =
  {northeast: LngLat, southwest: LngLat} |
  [LngLat, LngLat] |
  [number, number, number, number];
```

You can specify the bounds either using named corners, a pair of lat/lngs, or a four-tuple if you're confident you got the order right. Since `LngLat` already accommodates three forms, there are no fewer than 19 possible forms for `LngLatBounds`. Liberal indeed!

Now let's write a function that adjusts the viewport to accommodate a GeoJSON Feature and stores the new viewport in the URL (for a definition of `calculateBoundingBox`, see Item 31):

```
function focusOnFeature(f: Feature) {
  const bounds = calculateBoundingBox(f);
  const camera = viewportForBounds(bounds);
  setCamera(camera);
  const {center: {lat, lng}, zoom} = camera;
               // ~~~      Property 'lat' does not exist on type ...
               //      ~~~ Property 'lng' does not exist on type ...
  zoom;  // Type is number | undefined
  window.location.search = `?v=@${lat},${lng}z${zoom}`;
}
```

Whoops! Only the `zoom` property exists, but its type is inferred as `number|undefined`, which is also problematic. The issue is that the type declaration for `viewportForBounds` indicates that it is liberal not just in what it accepts but also in what it *produces*. The only type-safe way to use the `camera` result is to introduce a code branch for each component of the union type (Item 22).

The return type with lots of optional properties and union types makes viewportForBounds difficult to use. Its broad parameter type is convenient, but its broad return type is not. A more convenient API would be strict in what it produces.

One way to do this is to distinguish a canonical format for coordinates. Following JavaScript's convention of distinguishing "Array" and "Array-like" (Item 16), you can draw a distinction between LngLat and LngLatLike. You can also distinguish between a fully defined Camera type and the partial version accepted by setCamera:

```
interface LngLat { lng: number; lat: number; };
type LngLatLike = LngLat | { lon: number; lat: number; } | [number, number];

interface Camera {
  center: LngLat;
  zoom: number;
  bearing: number;
  pitch: number;
}
interface CameraOptions extends Omit<Partial<Camera>, 'center'> {
  center?: LngLatLike;
}
type LngLatBounds =
  {northeast: LngLatLike, southwest: LngLatLike} |
  [LngLatLike, LngLatLike] |
  [number, number, number, number];

declare function setCamera(camera: CameraOptions): void;
declare function viewportForBounds(bounds: LngLatBounds): Camera;
```

The loose CameraOptions type adapts the stricter Camera type (Item 14).

Using Partial<Camera> as the parameter type in setCamera would not work here since you do want to allow LngLatLike objects for the center property. And you can't write "CameraOptions extends Partial<Camera>" since LngLatLike is a superset of LngLat, not a subset (Item 7). If this seems too complicated, you could also write the type out explicitly at the cost of some repetition:

```
interface CameraOptions {
  center?: LngLatLike;
  zoom?: number;
  bearing?: number;
  pitch?: number;
}
```

In either case, with these new type declarations the focusOnFeature function passes the type checker:

```
function focusOnFeature(f: Feature) {
  const bounds = calculateBoundingBox(f);
  const camera = viewportForBounds(bounds);
  setCamera(camera);
  const {center: {lat, lng}, zoom} = camera;  // OK
  zoom;  // Type is number
  window.location.search = `?v=@${lat},${lng}z${zoom}`;
}
```

This time the type of `zoom` is `number`, rather than `number|undefined`. The `viewportForBounds` function is now much easier to use. If there were any other functions that produced bounds, you would also need to introduce a canonical form and a distinction between `LngLatBounds` and `LngLatBoundsLike`.

Is allowing 19 possible forms of bounding box a good design? Perhaps not. But if you're writing type declarations for a library that does this, you need to model its behavior. Just don't have 19 return types!

Things to Remember

- Input types tend to be broader than output types. Optional properties and union types are more common in parameter types than return types.
- To reuse types between parameters and return types, introduce a canonical form (for return types) and a looser form (for parameters).

Item 30: Don't Repeat Type Information in Documentation

What's wrong with this code?

```
/**
 * Returns a string with the foreground color.
 * Takes zero or one arguments. With no arguments, returns the
 * standard foreground color. With one argument, returns the foreground color
 * for a particular page.
 */
function getForegroundColor(page?: string) {
  return page === 'login' ? {r: 127, g: 127, b: 127} : {r: 0, g: 0, b: 0};
}
```

The code and the comment disagree! Without more context it's hard to say which is right, but something is clearly amiss. As a professor of mine used to say, "when your code and your comments disagree, they're both wrong!"

Let's assume that the code represents the desired behavior. There are a few issues with this comment:

- It says that the function returns the color as a `string` when it actually returns an `{r, g, b}` object.

- It explains that the function takes zero or one arguments, which is already clear from the type signature.
- It's needlessly wordy: the comment is longer than the function declaration *and* implementation!

TypeScript's type annotation system is designed to be compact, descriptive, and readable. Its developers are language experts with decades of experience. It's almost certainly a better way to express the types of your function's inputs and outputs than your prose!

And because your type annotations are checked by the TypeScript compiler, they'll never get out of sync with the implementation. Perhaps `getForegroundColor` used to return a string but was later changed to return an object. The person who made the change might have forgotten to update the long comment.

Nothing stays in sync unless it's forced to. With type annotations, TypeScript's type checker is that force! If you put type information in annotations and not in documentation, you greatly increase your confidence that it will remain correct as the code evolves.

A better comment might look like this:

```
/** Get the foreground color for the application or a specific page. */
function getForegroundColor(page?: string): Color {
  // ...
}
```

If you want to describe a particular parameter, use an `@param` JSDoc annotation. See Item 48 for more on this.

Comments about a lack of mutation are also suspect. Don't just say that you don't modify a parameter:

```
/** Does not modify nums */
function sort(nums: number[]) { /* ... */ }
```

Instead, declare it `readonly` (Item 17) and let TypeScript enforce the contract:

```
function sort(nums: readonly number[]) { /* ... */ }
```

What's true for comments is also true for variable names. Avoid putting types in them: rather than naming a variable `ageNum`, name it `age` and make sure it's really a `number`.

An exception to this is for numbers with units. If it's not clear what the units are, you may want to include them in a variable or property name. For instance, `timeMs` is a much clearer name than just `time`, and `temperatureC` is a much clearer name than `temperature`. Item 37 describes "brands," which provide a more type-safe approach to modeling units.

Things to Remember

- Avoid repeating type information in comments and variable names. In the best case it is duplicative of type declarations, and in the worst it will lead to conflicting information.
- Consider including units in variable names if they aren't clear from the type (e.g., `timeMs` or `temperatureC`).

Item 31: Push Null Values to the Perimeter of Your Types

When you first turn on `strictNullChecks`, it may seem as though you have to add scores of if statements checking for `null` and `undefined` values throughout your code. This is often because the relationships between null and non-null values are implicit: when variable A is non-null, you know that variable B is also non-null and vice versa. These implicit relationships are confusing both for human readers of your code and for the type checker.

Values are easier to work with when they're either completely null or completely non-null, rather than a mix. You can model this by pushing the null values out to the perimeter of your structures.

Suppose you want to calculate the min and max of a list of numbers. We'll call this the "extent." Here's an attempt:

```
function extent(nums: number[]) {
  let min, max;
  for (const num of nums) {
    if (!min) {
      min = num;
      max = num;
    } else {
      min = Math.min(min, num);
      max = Math.max(max, num);
    }
  }
  return [min, max];
}
```

The code type checks (without `strictNullChecks`) and has an inferred return type of `number[]`, which seems fine. But it has a bug and a design flaw:

- If the min or max is zero, it may get overridden. For example, `extent([0, 1, 2])` will return `[1, 2]` rather than `[0, 2]`.
- If the `nums` array is empty, the function will return `[undefined, undefined]`. This sort of object with several `undefined`s will be difficult for clients to work

with and is exactly the sort of type that this item discourages. We know from reading the source code that `min` and `max` will either both be `undefined` or neither, but that information isn't represented in the type system.

Turning on `strictNullChecks` makes both of these issues more apparent:

```
function extent(nums: number[]) {
  let min, max;
  for (const num of nums) {
    if (!min) {
      min = num;
      max = num;
    } else {
      min = Math.min(min, num);
      max = Math.max(max, num);
                  // ~~~ Argument of type 'number | undefined' is not
                  //     assignable to parameter of type 'number'
    }
  }
  return [min, max];
}
```

The return type of `extent` is now inferred as `(number | undefined)[]`, which makes the design flaw more apparent. This is likely to manifest as a type error wherever you call `extent`:

```
const [min, max] = extent([0, 1, 2]);
const span = max - min;
          // ~~~   ~~~ Object is possibly 'undefined'
```

The error in the implementation of `extent` comes about because you've excluded `undefined` as a value for `min` but not `max`. The two are initialized together, but this information isn't present in the type system. You could make it go away by adding a check for `max`, too, but this would be doubling down on the bug.

A better solution is to put the min and max in the same object and make this object either fully `null` or fully non-`null`:

```
function extent(nums: number[]) {
  let result: [number, number] | null = null;
  for (const num of nums) {
    if (!result) {
      result = [num, num];
    } else {
      result = [Math.min(num, result[0]), Math.max(num, result[1])];
    }
  }
  return result;
}
```

The return type is now `[number, number] | null`, which is easier for clients to work with. The min and max can be retrieved with either a non-null assertion:

```
const [min, max] = extent([0, 1, 2])!;
const span = max - min;  // OK
```

or a single check:

```
const range = extent([0, 1, 2]);
if (range) {
  const [min, max] = range;
  const span = max - min;  // OK
}
```

By using a single object to track the extent, we've improved our design, helped TypeScript understand the relationship between null values, and fixed the bug: the `if (!result)` check is now problem free.

A mix of null and non-null values can also lead to problems in classes. For instance, suppose you have a class that represents both a user and their posts on a forum:

```
class UserPosts {
  user: UserInfo | null;
  posts: Post[] | null;

  constructor() {
    this.user = null;
    this.posts = null;
  }

  async init(userId: string) {
    return Promise.all([
      async () => this.user = await fetchUser(userId),
      async () => this.posts = await fetchPostsForUser(userId)
    ]);
  }

  getUserName() {
    // ...?
  }
}
```

While the two network requests are loading, the `user` and `posts` properties will be `null`. At any time, they might both be `null`, one might be `null`, or they might both be non-`null`. There are four possibilities. This complexity will seep into every method on the class. This design is almost certain to lead to confusion, a proliferation of `null` checks, and bugs.

A better design would wait until all the data used by the class is available:

```
class UserPosts {
  user: UserInfo;
  posts: Post[];

  constructor(user: UserInfo, posts: Post[]) {
    this.user = user;
    this.posts = posts;
  }

  static async init(userId: string): Promise<UserPosts> {
    const [user, posts] = await Promise.all([
      fetchUser(userId),
      fetchPostsForUser(userId)
    ]);
    return new UserPosts(user, posts);
  }

  getUserName() {
    return this.user.name;
  }
}
```

Now the `UserPosts` class is fully non-`null`, and it's easy to write correct methods on it. Of course, if you need to perform operations while data is partially loaded, then you'll need to deal with the multiplicity of `null` and non-`null` states.

(Don't be tempted to replace nullable properties with Promises. This tends to lead to even more confusing code and forces all your methods to be async. Promises clarify the code that loads data but tend to have the opposite effect on the class that uses that data.)

Things to Remember

- Avoid designs in which one value being `null` or not `null` is implicitly related to another value being `null` or not `null`.
- Push `null` values to the perimeter of your API by making larger objects either `null` or fully non-`null`. This will make code clearer both for human readers and for the type checker.
- Consider creating a fully non-`null` class and constructing it when all values are available.
- While `strictNullChecks` may flag many issues in your code, it's indispensable for surfacing the behavior of functions with respect to null values.

Item 32: Prefer Unions of Interfaces to Interfaces of Unions

If you create an interface whose properties are union types, you should ask whether the type would make more sense as a union of more precise interfaces.

Suppose you're building a vector drawing program and want to define an interface for layers with specific geometry types:

```
interface Layer {
  layout: FillLayout | LineLayout | PointLayout;
  paint: FillPaint | LinePaint | PointPaint;
}
```

The `layout` field controls how and where the shapes are drawn (rounded corners? straight?), while the `paint` field controls styles (is the line blue? thick? thin? dashed?).

Would it make sense to have a layer whose `layout` is `LineLayout` but whose `paint` property is `FillPaint`? Probably not. Allowing this possibility makes using the library more error-prone and makes this interface difficult to work with.

A better way to model this is with separate interfaces for each type of layer:

```
interface FillLayer {
  layout: FillLayout;
  paint: FillPaint;
}
interface LineLayer {
  layout: LineLayout;
  paint: LinePaint;
}
interface PointLayer {
  layout: PointLayout;
  paint: PointPaint;
}
type Layer = FillLayer | LineLayer | PointLayer;
```

By defining `Layer` in this way, you've excluded the possibility of mixed `layout` and `paint` properties. This is an example of following Item 28's advice to prefer types that only represent valid states.

The most common example of this pattern is the "tagged union" (or "discriminated union"). In this case one of the properties is a union of string literal types:

```
interface Layer {
  type: 'fill' | 'line' | 'point';
  layout: FillLayout | LineLayout | PointLayout;
  paint: FillPaint | LinePaint | PointPaint;
}
```

As before, would it make sense to have `type: 'fill'` but then a `LineLayout` and `PointPaint`? Certainly not. Convert `Layer` to a union of interfaces to exclude this possibility:

```
interface FillLayer {
  type: 'fill';
  layout: FillLayout;
  paint: FillPaint;
}
interface LineLayer {
  type: 'line';
  layout: LineLayout;
  paint: LinePaint;
}
interface PointLayer {
  type: 'paint';
  layout: PointLayout;
  paint: PointPaint;
}
type Layer = FillLayer | LineLayer | PointLayer;
```

The `type` property is the "tag" and can be used to determine which type of `Layer` you're working with at runtime. TypeScript is also able to narrow the type of `Layer` based on the tag:

```
function drawLayer(layer: Layer) {
  if (layer.type === 'fill') {
    const {paint} = layer;  // Type is FillPaint
    const {layout} = layer;  // Type is FillLayout
  } else if (layer.type === 'line') {
    const {paint} = layer;  // Type is LinePaint
    const {layout} = layer;  // Type is LineLayout
  } else {
    const {paint} = layer;  // Type is PointPaint
    const {layout} = layer;  // Type is PointLayout
  }
}
```

By correctly modeling the relationship between the properties in this type, you help TypeScript check your code's correctness. The same code involving the initial `Layer` definition would have been cluttered with type assertions.

Because they work so well with TypeScript's type checker, tagged unions are ubiquitous in TypeScript code. Recognize this pattern and apply it when you can. If you can represent a data type in TypeScript with a tagged union, it's usually a good idea to do so. If you think of optional fields as a union of their type and `undefined`, then they fit this pattern as well. Consider this type:

```
interface Person {
  name: string;
  // These will either both be present or not be present
  placeOfBirth?: string;
  dateOfBirth?: Date;
}
```

The comment with type information is a strong sign that there might be a problem (Item 30). There is a relationship between the `placeOfBirth` and `dateOfBirth` fields that you haven't told TypeScript about.

A better way to model this is to move both of these properties into a single object. This is akin to moving `null` values to the perimeter (Item 31):

```
interface Person {
  name: string;
  birth?: {
    place: string;
    date: Date;
  }
}
```

Now TypeScript complains about values with a place but no date of birth:

```
const alanT: Person = {
  name: 'Alan Turing',
  birth: {
// ~~~~ Property 'date' is missing in type
//      '{ place: string; }' but required in type
//      '{ place: string; date: Date; }'
    place: 'London'
  }
}
```

Additionally, a function that takes a `Person` object only needs to do a single check:

```
function eulogize(p: Person) {
  console.log(p.name);
  const {birth} = p;
  if (birth) {
    console.log(`was born on ${birth.date} in ${birth.place}.`);
  }
}
```

If the structure of the type is outside your control (e.g., it's coming from an API), then you can still model the relationship between these fields using a now-familiar union of interfaces:

```
interface Name {
  name: string;
}

interface PersonWithBirth extends Name {
  placeOfBirth: string;
  dateOfBirth: Date;
}
```

```
type Person = Name | PersonWithBirth;
```

Now you get some of the same benefits as with the nested object:

```
function eulogize(p: Person) {
  if ('placeOfBirth' in p) {
    p // Type is PersonWithBirth
    const {dateOfBirth} = p  // OK, type is Date
  }
}
```

In both cases, the type definition makes the relationship between the properties more clear.

Things to Remember

- Interfaces with multiple properties that are union types are often a mistake because they obscure the relationships between these properties.
- Unions of interfaces are more precise and can be understood by TypeScript.
- Consider adding a "tag" to your structure to facilitate TypeScript's control flow analysis. Because they are so well supported, tagged unions are ubiquitous in TypeScript code.

Item 33: Prefer More Precise Alternatives to String Types

The domain of the `string` type is big: `"x"` and `"y"` are in it, but so is the complete text of *Moby Dick* (it starts `"Call me Ishmael…"` and is about 1.2 million characters long). When you declare a variable of type `string`, you should ask whether a narrower type would be more appropriate.

Suppose you're building a music collection and want to define a type for an album. Here's an attempt:

```
interface Album {
  artist: string;
  title: string;
  releaseDate: string;  // YYYY-MM-DD
  recordingType: string;  // E.g., "live" or "studio"
}
```

The prevalence of `string` types and the type information in comments (see Item 30) are strong indications that this `interface` isn't quite right. Here's what can go wrong:

```
const kindOfBlue: Album = {
  artist: 'Miles Davis',
  title: 'Kind of Blue',
  releaseDate: 'August 17th, 1959',  // Oops!
  recordingType: 'Studio',  // Oops!
};  // OK
```

The `releaseDate` field is incorrectly formatted (according to the comment) and `"Studio"` is capitalized where it should be lowercase. But these values *are* both strings, so this object is assignable to `Album` and the type checker doesn't complain.

These broad `string` types can mask errors for valid `Album` objects, too. For example:

```
function recordRelease(title: string, date: string) { /* ... */ }
recordRelease(kindOfBlue.releaseDate, kindOfBlue.title);  // OK, should be error
```

The parameters are reversed in the call to `recordRelease` but both are strings, so the type checker doesn't complain. Because of the prevalence of `string` types, code like this is sometimes called "stringly typed."

Can you make the types narrower to prevent these sorts of issues? While the complete text of *Moby Dick* would be a ponderous artist name or album title, it's at least plausible. So `string` is appropriate for these fields. For the `releaseDate` field it's better to just use a `Date` object and avoid issues around formatting. Finally, for the `recordingType` field, you can define a union type with just two values (you could also use an `enum`, but I generally recommend avoiding these; see Item 53):

```
type RecordingType = 'studio' | 'live';

interface Album {
  artist: string;
  title: string;
  releaseDate: Date;
  recordingType: RecordingType;
}
```

With these changes TypeScript is able to do a more thorough check for errors:

```
const kindOfBlue: Album = {
  artist: 'Miles Davis',
  title: 'Kind of Blue',
  releaseDate: new Date('1959-08-17'),
  recordingType: 'Studio'
// ~~~~~~~~~~~~ Type '"Studio"' is not assignable to type 'RecordingType'
};
```

There are advantages to this approach beyond stricter checking. First, explicitly defining the type ensures that its meaning won't get lost as it's passed around. If you wanted to find albums of just a certain recording type, for instance, you might define a function like this:

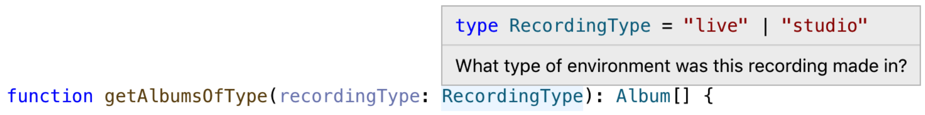
```
function getAlbumsOfType(recordingType: string): Album[] {
  // ...
}
```

How does the caller of this function know what `recordingType` is expected to be? It's just a `string`. The comment explaining that it's `"studio"` or `"live"` is hidden in the definition of `Album`, where the user might not think to look.

Second, explicitly defining a type allows you attach documentation to it (see Item 48):

```
/** What type of environment was this recording made in?  */
type RecordingType = 'live' | 'studio';
```

When you change `getAlbumsOfType` to take a `RecordingType`, the caller is able to click through and see the documentation (see Figure 4-1).



```
type RecordingType = "live" | "studio"
```

What type of environment was this recording made in?

```
function getAlbumsOfType(recordingType: RecordingType): Album[] {
```

Figure 4-1. Using a named type instead of string makes it possible to attach documentation to the type that is surfaced in your editor.

Another common misuse of `string` is in function parameters. Say you want to write a function that pulls out all the values for a single field in an array. The Underscore library calls this "pluck":

```
function pluck(records, key) {
  return record.map(record => record[key]);
}
```

How would you type this? Here's an initial attempt:

```
function pluck(record: any[], key: string): any[] {
  return record.map(r => r[key]);
}
```

This type checks but isn't great. The `any` types are problematic, particularly on the return value (see Item 38). The first step to improving the type signature is introducing a generic type parameter:

```
function pluck<T>(record: T[], key: string): any[] {
  return record.map(r => r[key]);
                      // ~~~~~~ Element implicitly has an 'any' type
                      //        because type '{}' has no index signature
}
```

TypeScript is now complaining that the `string` type for `key` is too broad. And it's right to do so: if you pass in an array of `Album`s then there are only four valid values for `key` ("artist," "title," "releaseDate," and "recordingType"), as opposed to the vast set of strings. This is precisely what the `keyof Album` type is:

```
type K = keyof Album;
// Type is "artist" | "title" | "releaseDate" | "recordingType"
```

So the fix is to replace `string` with `keyof T`:

```
function pluck<T>(record: T[], key: keyof T) {
  return record.map(r => r[key]);
}
```

This passes the type checker. We've also let TypeScript infer the return type. How does it do? If you mouse over `pluck` in your editor, the inferred type is:

```
function pluck<T>(record: T[], key: keyof T): T[keyof T][]
```

`T[keyof T]` is the type of any possible value in `T`. If you're passing in a single string as the `key`, this is too broad. For example:

```
const releaseDates = pluck(albums, 'releaseDate'); // Type is (string | Date)[]
```

The type should be `Date[]`, not `(string | Date)[]`. While `keyof T` is much narrower than `string`, it's *still* too broad. To narrow it further, we need to introduce a second generic parameter that is a subset of `keyof T` (probably a single value):

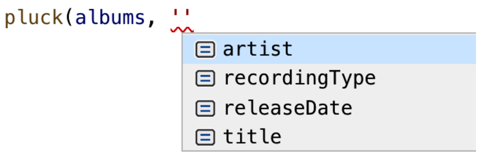
```
function pluck<T, K extends keyof T>(record: T[], key: K): T[K][] {
  return record.map(r => r[key]);
}
```

(For more on `extends` in this context, see Item 14.)

The type signature is now completely correct. We can check this by calling `pluck` in a few different ways:

```
pluck(albums, 'releaseDate'); // Type is Date[]
pluck(albums, 'artist');  // Type is string[]
pluck(albums, 'recordingType');  // Type is RecordingType[]
pluck(albums, 'recordingDate');
           // ~~~~~~~~~~~~~~~ Argument of type '"recordingDate"' is not
           //                 assignable to parameter of type ...
```

The language service is even able to offer autocomplete on the keys of `Album` (as shown in Figure 4-2).



```
pluck(albums, '')
```

Figure 4-2. Using a parameter type of keyof Album instead of string results in better autocomplete in your editor.

`string` has some of the same problems as `any`: when used inappropriately, it permits invalid values and hides relationships between types. This thwarts the type checker and can hide real bugs. TypeScript's ability to define subsets of `string` is a powerful way to bring type safety to JavaScript code. Using more precise types will both catch errors and improve the readability of your code.

Things to Remember

- Avoid "stringly typed" code. Prefer more appropriate types where not every `string` is a possibility.
- Prefer a union of string literal types to `string` if that more accurately describes the domain of a variable. You'll get stricter type checking and improve the development experience.
- Prefer `keyof T` to `string` for function parameters that are expected to be properties of an object.

Item 34: Prefer Incomplete Types to Inaccurate Types

In writing type declarations you'll inevitably find situations where you can model behavior in a more precise or less precise way. Precision in types is generally a good thing because it will help your users catch bugs and take advantage of the tooling that TypeScript provides. But take care as you increase the precision of your type declarations: it's easy to make mistakes, and incorrect types can be worse than no types at all.

Suppose you are writing type declarations for GeoJSON, a format we've seen before in Item 31. A GeoJSON Geometry can be one of a few types, each of which have differently shaped coordinate arrays:

```
interface Point {
  type: 'Point';
  coordinates: number[];
}
interface LineString {
  type: 'LineString';
  coordinates: number[][];
}
interface Polygon {
  type: 'Polygon';
  coordinates: number[][][];
}
type Geometry = Point | LineString | Polygon;  // Also several others
```

This is fine, but `number[]` for a coordinate is a bit imprecise. Really these are latitudes and longitudes, so perhaps a tuple type would be better:

```
type GeoPosition = [number, number];
interface Point {
  type: 'Point';
  coordinates: GeoPosition;
}
// Etc.
```

You publish your more precise types to the world and wait for the adulation to roll in. Unfortunately, a user complains that your new types have broken everything. Even though you've only ever used latitude and longitude, a position in GeoJSON is allowed to have a third element, an elevation, and potentially more. In an attempt to make the type declarations more precise, you've gone too far and made the types inaccurate! To continue using your type declarations, your user will have to introduce type assertions or silence the type checker entirely with `as any`.

As another example, consider trying to write type declarations for a Lisp-like language defined in JSON:

```
12
"red"
["+", 1, 2]  // 3
["/", 20, 2]  // 10
["case", [">", 20, 10], "red", "blue"]  // "red"
["rgb", 255, 0, 127]  // "#FF007F"
```

The Mapbox library uses a system like this to determine the appearance of map features across many devices. There's a whole spectrum of precision with which you could try to type this:

1. Allow anything.
2. Allow strings, numbers, and arrays.
3. Allow strings, numbers, and arrays starting with known function names.
4. Make sure each function gets the correct number of arguments.
5. Make sure each function gets the correct type of arguments.

The first two options are straightforward:

```
type Expression1 = any;
type Expression2 = number | string | any[];
```

Beyond this, you should introduce a test set of expressions that are valid and expressions that are not. As you make your types more precise, this will help prevent regressions (see Item 52):

```
const tests: Expression2[] = [
  10,
  "red",
  true,
// ~~~ Type 'true' is not assignable to type 'Expression2'
  ["+", 10, 5],
  ["case", [">", 20, 10], "red", "blue", "green"],  // Too many values
  ["**", 2, 31],  // Should be an error: no "**" function
  ["rgb", 255, 128, 64],
```

```
    ["rgb", 255, 0, 127, 0]  // Too many values
  ];
```

To go to the next level of precision you can use a union of string literal types as the first element of a tuple:

```
type FnName = '+' | '-' | '*' | '/' | '>' | '<' | 'case' | 'rgb';
type CallExpression = [FnName, ...any[]];
type Expression3 = number | string | CallExpression;

const tests: Expression3[] = [
  10,
  "red",
  true,
// ~~~ Type 'true' is not assignable to type 'Expression3'
  ["+", 10, 5],
  ["case", [">", 20, 10], "red", "blue", "green"],
  ["**", 2, 31],
// ~~~~~~~~~~~ Type '"**"' is not assignable to type 'FnName'
  ["rgb", 255, 128, 64]
];
```

There's one new caught error and no regressions. Pretty good!

What if you want to make sure that each function gets the correct number of arguments? This gets trickier since the type now needs to be recursive to reach down into all the function calls. As of TypeScript 3.6, to make this work you needed to introduce at least one `interface`. Since `interface`s can't be unions, you'll have to write the call expressions using `interface` instead. This is a bit awkward since fixed-length arrays are most easily expressed as tuple types. But you *can* do it:

```
type Expression4 = number | string | CallExpression;

type CallExpression = MathCall | CaseCall | RGBCall;

interface MathCall {
  0: '+' | '-' | '/' | '*' | '>' | '<';
  1: Expression4;
  2: Expression4;
  length: 3;
}

interface CaseCall {
  0: 'case';
  1: Expression4;
  2: Expression4;
  3: Expression4;
  length: 4 | 6 | 8 | 10 | 12 | 14 | 16 // etc.
}

interface RGBCall {
  0: 'rgb';
```

```
  1: Expression4;
  2: Expression4;
  3: Expression4;
  length: 4;
}

const tests: Expression4[] = [
  10,
  "red",
  true,
// ~~~ Type 'true' is not assignable to type 'Expression4'
  ["+", 10, 5],
  ["case", [">", 20, 10], "red", "blue", "green"],
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  Type '["case", [">", ...], ...]' is not assignable to type 'string'
  ["**", 2, 31],
// ~~~~~~~~~~~~ Type '["**", number, number]' is not assignable to type 'string
  ["rgb", 255, 128, 64],
  ["rgb", 255, 128, 64, 73]
// ~~~~~~~~~~~~~~~~~~~~~~~~ Type '["rgb", number, number, number, number]'
//                          is not assignable to type 'string'
];
```

Now all the invalid expressions produce errors. And it's interesting that you can express something like "an array of even length" using a TypeScript `interface`. But these error messages aren't very good, and the error about `**` has gotten quite a bit worse since the previous typings.

Is this an improvement over the previous, less precise types? The fact that you get errors for some incorrect usages is a win, but the errors will make this type more difficult to work with. Language services are as much a part of the TypeScript experience as type checking (see Item 6), so it's a good idea to look at the error messages resulting from your type declarations and try autocomplete in situations where it should work. If your new type declarations are more precise but break autocomplete, then they'll make for a less enjoyable TypeScript development experience.

The complexity of this type declaration has also increased the odds that a bug will creep in. For example, `Expression4` requires that all math operators take two parameters, but the Mapbox expression spec says that `+` and `*` can take more. Also, `-` can take a single parameter, in which case it negates its input. `Expression4` incorrectly flags errors in all of these:

```
 const okExpressions: Expression4[] = [
   ['-', 12],
// ~~~~~~~~~ Type '["-", number]' is not assignable to type 'string'
   ['+', 1, 2, 3],
// ~~~~~~~~~~~~~~ Type '["+", number, ...]' is not assignable to type 'string'
   ['*', 2, 3, 4],
// ~~~~~~~~~~~~~~ Type '["*", number, ...]' is not assignable to type 'string'
 ];
```

Once again, in trying to be more precise we've overshot and become inaccurate. These inaccuracies can be corrected, but you'll want to expand your test set to convince yourself that you haven't missed anything else. Complex code generally requires more tests, and the same is true of types.

As you refine types, it can be helpful to think of the "uncanny valley" metaphor. Refining very imprecise types like `any` is usually helpful. But as your types get more precise, the expectation that they'll also be accurate increases. You'll start to rely on the types more, and so inaccuracies will produce bigger problems.

Things to Remember

- Avoid the uncanny valley of type safety: incorrect types are often worse than no types.
- If you cannot model a type accurately, do not model it inaccurately! Acknowledge the gaps using `any` or `unknown`.
- Pay attention to error messages and autocomplete as you make typings increasingly precise. It's not just about correctness: developer experience matters, too.

Item 35: Generate Types from APIs and Specs, Not Data

The other items in this chapter have discussed the many benefits of designing your types well and shown what can go wrong if you don't. A well-designed type makes TypeScript a pleasure to use, while a poorly designed one can make it miserable. But this does put quite a bit of pressure on type design. Wouldn't it be nice if you didn't have to do this yourself?

At least some of your types are likely to come from outside your program: file formats, APIs, or specs. In these cases you may be able to avoid writing types by generating them instead. If you do this, the key is to generate types from specifications, rather than from example data. When you generate types from a spec, TypeScript will help ensure that you haven't missed any cases. When you generate types from data, you're only considering the examples you've seen. You might be missing important edge cases that could break your program.

In Item 31 we wrote a function to calculate the bounding box of a GeoJSON Feature. Here's what it looked like:

```
function calculateBoundingBox(f: GeoJSONFeature): BoundingBox | null {
  let box: BoundingBox | null = null;

  const helper = (coords: any[]) => {
    // ...
  };
  const {geometry} = f;
  if (geometry) {
    helper(geometry.coordinates);
  }

  return box;
}
```

The `GeoJSONFeature` type was never explicitly defined. You could write it using some of the examples from Item 31. But a better approach is to use the formal GeoJSON spec.[1] Fortunately for us, there are already TypeScript type declarations for it on DefinitelyTyped. You can add these in the usual way:

```
$ npm install --save-dev @types/geojson
+ @types/geojson@7946.0.7
```

When you plug in the GeoJSON declarations, TypeScript immediately flags an error:

```
import {Feature} from 'geojson';

function calculateBoundingBox(f: Feature): BoundingBox | null {
  let box: BoundingBox | null = null;

  const helper = (coords: any[]) => {
    // ...
  };

  const {geometry} = f;
  if (geometry) {
    helper(geometry.coordinates);
                // ~~~~~~~~~~~
                // Property 'coordinates' does not exist on type 'Geometry'
                //   Property 'coordinates' does not exist on type
                //   'GeometryCollection'
  }

  return box;
}
```

The problem is that your code assumes a geometry will have a `coordinates` property. This is true for many geometries, including points, lines, and polygons. But a GeoJSON geometry can also be a `GeometryCollection`, a heterogeneous collection of other geometries. Unlike the other geometry types, it does not have a `coordinates` property.

1 GeoJSON is also known as RFC 7946. The very readable spec is at *http://geojson.org*.

If you call `calculateBoundingBox` on a Feature whose geometry is a `GeometryCollection`, it will throw an error about not being able to read property `0` of `undefined`. This is a real bug! And we caught it using type definitions from a spec.

One option for fixing it is to explicitly disallow `GeometryCollections`, as shown here:

```
const {geometry} = f;
if (geometry) {
  if (geometry.type === 'GeometryCollection') {
    throw new Error('GeometryCollections are not supported.');
  }
  helper(geometry.coordinates);  // OK
}
```

TypeScript is able to refine the type of `geometry` based on the check, so the reference to `geometry.coordinates` is allowed. If nothing else, this results in a clearer error message for the user.

But the better solution is to support all the types of geometry! You can do this by pulling out another helper function:

```
const geometryHelper = (g: Geometry) => {
  if (geometry.type === 'GeometryCollection') {
    geometry.geometries.forEach(geometryHelper);
  } else {
    helper(geometry.coordinates);  // OK
  }
}

const {geometry} = f;
if (geometry) {
  geometryHelper(geometry);
}
```

Had you written type declarations for GeoJSON yourself, you would have based them off of your understanding and experience with the format. This might not have included `GeometryCollections` and would have led to a false sense of security about your code's correctness. Using types based on a spec gives you confidence that your code will work with all values, not just the ones you've seen.

Similar considerations apply to API calls: if you can generate types from the specification of an API, then it is usually a good idea to do so. This works particularly well with APIs that are typed themselves, such as GraphQL.

A GraphQL API comes with a schema that specifies all the possible queries and interfaces using a type system somewhat similar to TypeScript. You write queries that request specific fields in these interfaces. For example, to get information about a repository using the GitHub GraphQL API you might write:

```
query {
  repository(owner: "Microsoft", name: "TypeScript") {
    createdAt
    description
  }
}
```

The result is:

```
  {
    "data": {
      "repository": {
        "createdAt": "2014-06-17T15:28:39Z",
        "description":
          "TypeScript is a superset of JavaScript that compiles to JavaScript."
      }
    }
  }
```

The beauty of this approach is that you can generate TypeScript types *for your specific query*. As with the GeoJSON example, this helps ensure that you model the relationships between types and their nullability accurately.

Here's a query to get the open source license for a GitHub repository:

```
query getLicense($owner:String!, $name:String!){
  repository(owner:$owner, name:$name) {
    description
    licenseInfo {
      spdxId
      name
    }
  }
}
```

`$owner` and `$name` are GraphQL variables which are themselves typed. The type syntax is similar enough to TypeScript that it can be confusing to go back and forth. `String` is a GraphQL type—it would be `string` in TypeScript (see Item 10). And while TypeScript types are not nullable, types in GraphQL are. The `!` after the type indicates that it is guaranteed to not be null.

There are many tools to help you go from a GraphQL query to TypeScript types. One is Apollo. Here's how you use it:

```
$ apollo client:codegen \
    --endpoint https://api.github.com/graphql \
    --includes license.graphql \
    --target typescript
Loading Apollo Project
Generating query files with 'typescript' target - wrote 2 files
```

You need a GraphQL schema to generate types for a query. Apollo gets this from the `api.github.com/graphql` endpoint. The output looks like this:

```
export interface getLicense_repository_licenseInfo {
  __typename: "License";
  /** Short identifier specified by <https://spdx.org/licenses> */
  spdxId: string | null;
  /** The license full name specified by <https://spdx.org/licenses> */
  name: string;
}

export interface getLicense_repository {
  __typename: "Repository";
  /** The description of the repository. */
  description: string | null;
  /** The license associated with the repository */
  licenseInfo: getLicense_repository_licenseInfo | null;
}

export interface getLicense {
  /** Lookup a given repository by the owner and repository name. */
  repository: getLicense_repository | null;
}

export interface getLicenseVariables {
  owner: string;
  name: string;
}
```

The important bits to note here are that:

- Interfaces are generated for both the query parameters (`getLicenseVariables`) and the response (`getLicense`).
- Nullability information is transferred from the schema to the response interfaces. The `repository`, `description`, `licenseInfo`, and `spdxId` fields are nullable, whereas the license `name` and the query variables are not.
- Documentation is transferred as JSDoc so that it appears in your editor (Item 48). These comments come from the GraphQL schema itself.

This type information helps ensure that you use the API correctly. If your queries change, the types will change. If the schema changes, then so will your types. There is no risk that your types and reality diverge since they are both coming from a single source of truth: the GraphQL schema.

What if there's no spec or official schema available? Then you'll have to generate types from data. Tools like `quicktype` can help with this. But be aware that your types may not match reality: there may be edge cases that you've missed.

Even if you're not aware of it, you are already benefiting from code generation. Type-Script's type declarations for the browser DOM API are generated from the official interfaces (see Item 55). This ensures that they correctly model a complicated system and helps TypeScript catch errors and misunderstandings in your own code.

Things to Remember

- Consider generating types for API calls and data formats to get type safety all the way to the edge of your code.
- Prefer generating code from specs rather than data. Rare cases matter!

Item 36: Name Types Using the Language of Your Problem Domain

> There are only two hard problems in Computer Science: cache invalidation and naming things.
>
> —Phil Karlton

This book has had much to say about the *shape* of types and the sets of values in their domains, but much less about what you *name* your types. But this is an important part of type design, too. Well-chosen type, property, and variable names can clarify intent and raise the level of abstraction of your code and types. Poorly chosen types can obscure your code and lead to incorrect mental models.

Suppose you're building out a database of animals. You create an interface to represent one:

```
interface Animal {
  name: string;
  endangered: boolean;
  habitat: string;
}

const leopard: Animal = {
  name: 'Snow Leopard',
  endangered: false,
  habitat: 'tundra',
};
```

There are a few issues here:

- `name` is a very general term. What sort of name are you expecting? A scientific name? A common name?
- The boolean `endangered` field is also ambiguous. What if an animal is extinct? Is the intent here "endangered or worse?" Or does it literally mean endangered?

- The `habitat` field is very ambiguous, not just because of the overly broad `string` type (Item 33) but also because it's unclear what's meant by "habitat."
- The variable name is `leopard`, but the value of the `name` property is "Snow Leopard." Is this distinction meaningful?

Here's a type declaration and value with less ambiguity:

```
interface Animal {
  commonName: string;
  genus: string;
  species: string;
  status: ConservationStatus;
  climates: KoppenClimate[];
}
type ConservationStatus = 'EX' | 'EW' | 'CR' | 'EN' | 'VU' | 'NT' | 'LC';
type KoppenClimate = |
  'Af' | 'Am' | 'As' | 'Aw' |
  'BSh' | 'BSk' | 'BWh' | 'BWk' |
  'Cfa' | 'Cfb' | 'Cfc' | 'Csa' | 'Csb' | 'Csc' | 'Cwa' | 'Cwb' | 'Cwc' |
  'Dfa' | 'Dfb' | 'Dfc' | 'Dfd' |
  'Dsa' | 'Dsb' | 'Dsc' | 'Dwa' | 'Dwb' | 'Dwc' | 'Dwd' |
  'EF' | 'ET';
const snowLeopard: Animal = {
  commonName: 'Snow Leopard',
  genus: 'Panthera',
  species: 'Uncia',
  status: 'VU',  // vulnerable
  climates: ['ET', 'EF', 'Dfd'],  // alpine or subalpine
};
```

This makes a number of improvements:

- `name` has been replaced with more specific terms: `commonName`, `genus`, and `species`.
- `endangered` has become `conservationStatus` and uses a standard classification system from the IUCN.
- `habitat` has become `climates` and uses another standard taxonomy, the Köppen climate classification.

If you needed more information about the fields in the first version of this type, you'd have to go find the person who wrote them and ask. In all likelihood, they've left the company or don't remember. Worse yet, you might run `git blame` to find out who wrote these lousy types, only to find that it was you!

The situation is much improved with the second version. If you want to learn more about the Köppen climate classification system or track down what the precise meaning of a conservation status is, then there are myriad resources online to help you.

Every domain has specialized vocabulary to describe its subject. Rather than inventing your own terms, try to reuse terms from the domain of your problem. These vocabularies have often been honed over years, decades, or centuries and are well understood by people in the field. Using these terms will help you communicate with users and increase the clarity of your types.

Take care to use domain vocabulary accurately: co-opting the language of a domain to mean something different is even more confusing than inventing your own.

Here are a few other rules to keep in mind as you name types, properties, and variables:

- Make distinctions meaningful. In writing and speech it can be tedious to use the same word over and over. We introduce synonyms to break the monotony. This makes prose more enjoyable to read, but it has the opposite effect on code. If you use two different terms, make sure you're drawing a meaningful distinction. If not, you should use the same term.
- Avoid vague, meaningless names like "data," "info," "thing," "item," "object," or the ever-popular "entity." If Entity has a specific meaning in your domain, fine. But if you're using it because you don't want to think of a more meaningful name, then you'll eventually run into trouble.
- Name things for what they are, not for what they contain or how they are computed. `Directory` is more meaningful than `INodeList`. It allows you to think about a directory as a concept, rather than in terms of its implementation. Good names can increase your level of abstraction and decrease your risk of inadvertent collisions.

Things to Remember

- Reuse names from the domain of your problem where possible to increase the readability and level of abstraction of your code.
- Avoid using different names for the same thing: make distinctions in names meaningful.

Item 37: Consider "Brands" for Nominal Typing

Item 4 discussed structural ("duck") typing and how it can sometimes lead to surprising results:

```
interface Vector2D {
  x: number;
  y: number;
}
function calculateNorm(p: Vector2D) {
  return Math.sqrt(p.x * p.x + p.y * p.y);
}

calculateNorm({x: 3, y: 4});  // OK, result is 5
const vec3D = {x: 3, y: 4, z: 1};
calculateNorm(vec3D);  // OK! result is also 5
```

What if you'd like `calculateNorm` to reject 3D vectors? This goes against the structural typing model of TypeScript but is certainly more mathematically correct.

One way to achieve this is with *nominal typing*. With nominal typing, a value is a `Vector2D` because you say it is, not because it has the right shape. To approximate this in TypeScript, you can introduce a "brand" (think cows, not Coca-Cola):

```
interface Vector2D {
  _brand: '2d';
  x: number;
  y: number;
}
function vec2D(x: number, y: number): Vector2D {
  return {x, y, _brand: '2d'};
}
function calculateNorm(p: Vector2D) {
  return Math.sqrt(p.x * p.x + p.y * p.y);  // Same as before
}

calculateNorm(vec2D(3, 4)); // OK, returns 5
const vec3D = {x: 3, y: 4, z: 1};
calculateNorm(vec3D);
           // ~~~~~ Property '_brand' is missing in type...
```

The brand ensures that the vector came from the right place. Granted there's nothing stopping you from adding `_brand: '2d'` to the `vec3D` value. But this is moving from the accidental into the malicious. This sort of brand is typically enough to catch inadvertent misuses of functions.

Interestingly, you can get many of the same benefits as explicit brands while operating only in the type system. This removes runtime overhead and also lets you brand built-in types like `string` or `number` where you can't attach additional properties.

For instance, what if you have a function that operates on the filesystem and requires an absolute (as opposed to a relative) path? This is easy to check at runtime (does the path start with "/"?) but not so easy in the type system.

Here's an approach with brands:

```
type AbsolutePath = string & {_brand: 'abs'};
function listAbsolutePath(path: AbsolutePath) {
  // ...
}
```

```
function isAbsolutePath(path: string): path is AbsolutePath {
  return path.startsWith('/');
}
```

You can't construct an object that is a `string` and has a `_brand` property. This is purely a game with the type system.

If you have a `string` path that could be either absolute or relative, you can check using the type guard, which will refine its type:

```
function f(path: string) {
  if (isAbsolutePath(path)) {
    listAbsolutePath(path);
  }
  listAbsolutePath(path);
                // ~~~~ Argument of type 'string' is not assignable
                //      to parameter of type 'AbsolutePath'
}
```

This sort of approach could be helpful in documenting which functions expect absolute or relative paths and which type of path each variable holds. It is not an ironclad guarantee, though: `path as AbsolutePath` will succeed for any `string`. But if you avoid these sorts of assertions, then the only way to get an `AbsolutePath` is to be given one or to check, which is exactly what you want.

This approach can be used to model many properties that cannot be expressed within the type system. For example, using binary search to find an element in a list:

```
function binarySearch<T>(xs: T[], x: T): boolean {
  let low = 0, high = xs.length - 1;
  while (high >= low) {
    const mid = low + Math.floor((high - low) / 2);
    const v = xs[mid];
    if (v === x) return true;
    [low, high] = x > v ? [mid + 1, high] : [low, mid - 1];
  }
  return false;
}
```

This works if the list is sorted, but will result in false negatives if it is not. You can't represent a sorted list in TypeScript's type system. But you can create a brand:

```
type SortedList<T> = T[] & {_brand: 'sorted'};

function isSorted<T>(xs: T[]): xs is SortedList<T> {
  for (let i = 1; i < xs.length; i++) {
    if (xs[i] > xs[i - 1]) {
      return false;
    }
  }
  return true;
}
```

```
function binarySearch<T>(xs: SortedList<T>, x: T): boolean {
  // ...
}
```

In order to call this version of `binarySearch`, you either need to be given a `SortedList` (i.e., have a proof that the list is sorted) or prove that it's sorted yourself using `isSorted`. The linear scan isn't great, but at least you'll be safe!

This is a helpful perspective to have on the type checker in general. In order to call a method on an object, for instance, you either need to be given a non-`null` object or prove that it's non-`null` yourself with a conditional.

You can also brand `number` types—for example, to attach units:

```
type Meters = number & {_brand: 'meters'};
type Seconds = number & {_brand: 'seconds'};

const meters = (m: number) => m as Meters;
const seconds = (s: number) => s as Seconds;

const oneKm = meters(1000);  // Type is Meters
const oneMin = seconds(60);  // Type is Seconds
```

This can be awkward in practice since arithmetic operations make the numbers forget their brands:

```
const tenKm = oneKm * 10;  // Type is number
const v = oneKm / oneMin;  // Type is number
```

If your code involves lots of numbers with mixed units, however, this may still be an attractive approach to documenting the expected types of numeric parameters.

Things to Remember

- TypeScript uses structural ("duck") typing, which can sometimes lead to surprising results. If you need nominal typing, consider attaching "brands" to your values to distinguish them.
- In some cases you may be able to attach brands entirely in the type system, rather than at runtime. You can use this technique to model properties outside of TypeScript's type system.

CHAPTER 5

Working with any

Type systems were traditionally binary affairs: either a language had a fully static type system or a fully dynamic one. TypeScript blurs the line, because its type system is *optional* and *gradual*. You can add types to parts of your program but not others.

This is essential for migrating existing JavaScript codebases to TypeScript bit by bit (Chapter 8). Key to this is the `any` type, which effectively disables type checking for parts of your code. It is both powerful and prone to abuse. Learning to use `any` wisely is essential for writing effective TypeScript. This chapter walks you through how to limit the downsides of `any` while still retaining its benefits.

Item 38: Use the Narrowest Possible Scope for any Types

Consider this code:

```
function processBar(b: Bar) { /* ... */ }

function f() {
  const x = expressionReturningFoo();
  processBar(x);
  //         ~ Argument of type 'Foo' is not assignable to
  //           parameter of type 'Bar'
}
```

If you somehow know from context that `x` is assignable to `Bar` in addition to `Foo`, you can force TypeScript to accept this code in two ways:

```
function f1() {
  const x: any = expressionReturningFoo();  // Don't do this
  processBar(x);
}

function f2() {
  const x = expressionReturningFoo();
  processBar(x as any);  // Prefer this
}
```

Of these, the second form is vastly preferable. Why? Because the any type is scoped to a single expression in a function argument. It has no effect outside this argument or this line. If code after the processBar call references x, its type will still be Foo, and it will still be able to trigger type errors, whereas in the first example its type is any until it goes out of scope at the end of the function.

The stakes become significantly higher if you *return* x from this function. Look what happens:

```
  function f1() {
    const x: any = expressionReturningFoo();
    processBar(x);
    return x;
  }

  function g() {
    const foo = f1();  // Type is any
    foo.fooMethod();  // This call is unchecked!
  }
```

An any return type is "contagious" in that it can spread throughout a codebase. As a result of our changes to f, an any type has quietly appeared in g. This would not have happened with the more narrowly scoped any in f2.

(This is a good reason to consider including explicit return type annotations, even when the return type can be inferred. It prevents an any type from "escaping." See discussion in Item 19.)

We used any here to silence an error that we believed to be incorrect. Another way to do this is with @ts-ignore:

```
  function f1() {
    const x = expressionReturningFoo();
    // @ts-ignore
    processBar(x);
    return x;
  }
```

This silences an error on the next line, leaving the type of x unchanged. Try not to lean too heavily on @ts-ignore: the type checker usually has a good reason to complain. It also means that if the error on the next line changes to something more problematic, you won't know.

You may also run into situations where you get a type error for just one property in a larger object:

```
const config: Config = {
  a: 1,
  b: 2,
  c: {
    key: value
 // ~~~ Property ... missing in type 'Bar' but required in type 'Foo'
  }
};
```

You can silence errors like this by throwing an `as any` around the whole `config` object:

```
const config: Config = {
  a: 1,
  b: 2,
  c: {
    key: value
  }
} as any;  // Don't do this!
```

But this has the side effect of disabling type checking for the other properties (`a` and `b`) as well. Using a more narrowly scoped `any` limits the damage:

```
const config: Config = {
  a: 1,
  b: 2,  // These properties are still checked
  c: {
    key: value as any
  }
};
```

Things to Remember

- Make your uses of `any` as narrowly scoped as possible to avoid undesired loss of type safety elsewhere in your code.
- Never return an `any` type from a function. This will silently lead to the loss of type safety for any client calling the function.
- Consider `@ts-ignore` as an alternative to `any` if you need to silence one error.

Item 39: Prefer More Precise Variants of any to Plain any

The `any` type encompasses all values that can be expressed in JavaScript. This is a vast set! It includes not just all numbers and strings, but all arrays, objects, regular expressions, functions, classes, and DOM elements, not to mention `null` and `undefined`. When you use an `any` type, ask whether you really had something more specific in mind. Would it be OK to pass in a regular expression or a function?

Often the answer is “no,” in which case you might be able to retain some type safety by using a more specific type:

```
function getLengthBad(array: any) {  // Don't do this!
  return array.length;
}

function getLength(array: any[]) {
  return array.length;
}
```

The latter version, which uses `any[]` instead of `any`, is better in three ways:

- The reference to `array.length` in the function body is type checked.
- The function’s return type is inferred as `number` instead of `any`.
- Calls to `getLength` will be checked to ensure that the parameter is an array:

```
getLengthBad(/123/);  // No error, returns undefined
getLength(/123/);
       // ~~~~~ Argument of type 'RegExp' is not assignable
       //       to parameter of type 'any[]'
```

If you expect a parameter to be an array of arrays but don’t care about the type, you can use `any[][]`. If you expect some sort of object but don’t know what the values will be, you can use `{[key: string]: any}`:

```
function hasTwelveLetterKey(o: {[key: string]: any}) {
  for (const key in o) {
    if (key.length === 12) {
      return true;
    }
  }
  return false;
}
```

You could also use the `object` type in this situation, which includes all non-primitive types. This is slightly different in that, while you can still enumerate keys, you can’t access the values of any of them:

```
function hasTwelveLetterKey(o: object) {
  for (const key in o) {
    if (key.length === 12) {
      console.log(key, o[key]);
                  //  ~~~~~~ Element implicitly has an 'any' type
                  //         because type '{}' has no index signature
      return true;
    }
  }
  return false;
}
```

If this sort of type fits your needs, you might also be interested in the `unknown` type. See Item 42.

Avoid using `any` if you expect a function type. You have several options here depending on how specific you want to get:

```
type Fn0 = () => any;  // any function callable with no params
type Fn1 = (arg: any) => any;  // With one param
type FnN = (...args: any[]) => any;  // With any number of params
                                     // same as "Function" type
```

All of these are more precise than `any` and hence preferable to it. Note the use of `any[]` as the type for the rest parameter in the last example. `any` would also work here but would be less precise:

```
const numArgsBad = (...args: any) => args.length; // Returns any
const numArgsGood = (...args: any[]) => args.length;  // Returns number
```

This is perhaps the most common use of the `any[]` type.

Things to Remember

- When you use `any`, think about whether any JavaScript value is truly permissible.
- Prefer more precise forms of `any` such as `any[]` or `{[id: string]: any}` or `() => any` if they more accurately model your data.

Item 40: Hide Unsafe Type Assertions in Well-Typed Functions

There are many functions whose type signatures are easy to write but whose implementations are quite difficult to write in type-safe code. And while writing type-safe implementations is a noble goal, it may not be worth the difficulty to deal with edge cases that you know don't come up in your code. If a reasonable attempt at a type-safe implementation doesn't work, use an unsafe type assertion hidden inside a function with the right type signature. Unsafe assertions hidden inside well-typed functions are much better than unsafe assertions scattered throughout your code.

Suppose you want to make a function cache its last call. This is a common technique for eliminating expensive function calls with frameworks like React.[1] It would be nice to write a general `cacheLast` wrapper that adds this behavior to any function. Its declaration is easy to write:

1 If you are using React, you should use the built-in `useMemo` hook, rather than rolling your own.

```
declare function cacheLast<T extends Function>(fn: T): T;
```

Here's an attempt at an implementation:

```
function cacheLast<T extends Function>(fn: T): T {
  let lastArgs: any[]|null = null;
  let lastResult: any;
  return function(...args: any[]) {
      // ~~~~~~~~~~~~~~~~~~~~~~~~~~
      //          Type '(...args: any[]) => any' is not assignable to type 'T'
    if (!lastArgs || !shallowEqual(lastArgs, args)) {
      lastResult = fn(...args);
      lastArgs = args;
    }
    return lastResult;
  };
}
```

The error makes sense: TypeScript has no reason to believe that this very loose function has any relation to `T`. But you know that the type system will enforce that it's called with the right parameters and that its return value is given the correct type. So you shouldn't expect too many problems if you add a type assertion here:

```
function cacheLast<T extends Function>(fn: T): T {
  let lastArgs: any[]|null = null;
  let lastResult: any;
  return function(...args: any[]) {
    if (!lastArgs || !shallowEqual(lastArgs, args)) {
      lastResult = fn(...args);
      lastArgs = args;
    }
    return lastResult;
  } as unknown as T;
}
```

And indeed this will work great for any simple function you pass it. There are quite a few `any` types hidden in this implementation, but you've kept them out of the type signature, so the code that calls `cacheLast` will be none the wiser.

(Is this actually safe? There are a few real problems with this implementation: it doesn't check that the values of `this` for successive calls are the same. And if the original function had properties defined on it, then the wrapped function would not have these, so it wouldn't have the same type. But if you know that these situations don't come up in your code, this implementation is just fine. This function *can* be written in a type-safe way, but it is a more complex exercise that is left to the reader.)

The `shallowEqual` function from the previous example operated on two arrays and is easy to type and implement. But the object variation is more interesting. As with `cacheLast`, it's easy to write its type signature:

```
declare function shallowObjectEqual<T extends object>(a: T, b: T): boolean;
```

The implementation requires some care since there's no guarantee that a and b have the same keys (see Item 54):

```
function shallowObjectEqual<T extends object>(a: T, b: T): boolean {
  for (const [k, aVal] of Object.entries(a)) {
    if (!(k in b) || aVal !== b[k]) {
                          // ~~~~ Element implicitly has an 'any' type
                          //      because type '{}' has no index signature
      return false;
    }
  }
  return Object.keys(a).length === Object.keys(b).length;
}
```

It's a bit surprising that TypeScript complains about the b[k] access despite your having just checked that k in b is true. But it does, so you have no choice but to cast:

```
function shallowObjectEqual<T extends object>(a: T, b: T): boolean {
  for (const [k, aVal] of Object.entries(a)) {
    if (!(k in b) || aVal !== (b as any)[k]) {
      return false;
    }
  }
  return Object.keys(a).length === Object.keys(b).length;
}
```

This type assertion is harmless (since you've checked k in b), and you're left with a correct function with a clear type signature. This is much preferable to scattering iteration and assertions to check for object equality throughout your code!

Things to Remember

- Sometimes unsafe type assertions are necessary or expedient. When you need to use one, hide it inside a function with a correct signature.

Item 41: Understand Evolving any

In TypeScript a variable's type is generally determined when it is declared. After this, it can be *refined* (by checking if it is null, for instance), but it cannot expand to include new values. There is one notable exception to this, however, involving any types.

In JavaScript, you might write a function to generate a range of numbers like this:

```
function range(start, limit) {
  const out = [];
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
```

```
  return out;
}
```

When you convert this to TypeScript, it works exactly as you'd expect:

```
function range(start: number, limit: number) {
  const out = [];
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
  return out;  // Return type inferred as number[]
}
```

Upon closer inspection, however, it's surprising that this works! How does TypeScript know that the type of `out` is `number[]` when it's initialized as `[]`, which could be an array of any type?

Inspecting each of the three occurrences of `out` to reveal its inferred type starts to tell the story:

```
function range(start: number, limit: number) {
  const out = [];  // Type is any[]
  for (let i = start; i < limit; i++) {
    out.push(i);  // Type of out is any[]
  }
  return out;  // Type is number[]
}
```

The type of `out` starts as `any[]`, an undifferentiated array. But as we push `number` values onto it, its type "evolves" to become `number[]`.

This is distinct from narrowing (Item 22). An array's type can expand by pushing different elements onto it:

```
const result = [];  // Type is any[]
result.push('a');
result  // Type is string[]
result.push(1);
result  // Type is (string | number)[]
```

With conditionals, the type can even vary across branches. Here we show the same behavior with a simple value, rather than an array:

```
let val;  // Type is any
if (Math.random() < 0.5) {
  val = /hello/;
  val  // Type is RegExp
} else {
  val = 12;
  val  // Type is number
}
val  // Type is number | RegExp
```

A final case that triggers this "evolving any" behavior is if a variable is initially `null`. This often comes up when you set a value in a `try`/`catch` block:

```
let val = null;  // Type is any
try {
  somethingDangerous();
  val = 12;
  val  // Type is number
} catch (e) {
  console.warn('alas!');
}
val  // Type is number | null
```

Interestingly, this behavior only happens when a variable's type is implicitly `any` with `noImplicitAny` set! Adding an *explicit* `any` keeps the type constant:

```
let val: any;  // Type is any
if (Math.random() < 0.5) {
  val = /hello/;
  val  // Type is any
} else {
  val = 12;
  val  // Type is any
}
val  // Type is any
```

This behavior can be confusing to follow in your editor since the type is only "evolved" *after* you assign or push an element. Inspecting the type on the line with the assignment will still show `any` or `any[]`.

If you use a value before any assignment to it, you'll get an implicit any error:

```
function range(start: number, limit: number) {
  const out = [];
  //    ~~~ Variable 'out' implicitly has type 'any[]' in some
  //        locations where its type cannot be determined
  if (start === limit) {
    return out;
    //     ~~~ Variable 'out' implicitly has an 'any[]' type
  }
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
  return out;
}
```

Put another way, "evolving" `any` types are only `any` when you *write* to them. If you try to *read* from them while they're still `any`, you'll get an error.

Implicit `any` types do not evolve through function calls. The arrow function here trips up inference:

```
function makeSquares(start: number, limit: number) {
  const out = [];
     // ~~~ Variable 'out' implicitly has type 'any[]' in some locations
  range(start, limit).forEach(i => {
    out.push(i * i);
  });
  return out;
      // ~~~ Variable 'out' implicitly has an 'any[]' type
}
```

In cases like this, you may want to consider using an array's `map` and `filter` methods to build arrays in a single statement and avoid iteration and evolving `any` entirely. See Items 23 and 27.

Evolving `any` comes with all the usual caveats about type inference. Is the correct type for your array really `(string|number)[]`? Or should it be `number[]` and you incorrectly pushed a `string`? You may still want to provide an explicit type annotation to get better error checking instead of using evolving `any`.

Things to Remember

- While TypeScript types typically only *refine*, implicit `any` and `any[]` types are allowed to *evolve*. You should be able to recognize and understand this construct where it occurs.
- For better error checking, consider providing an explicit type annotation instead of using evolving `any`.

Item 42: Use unknown Instead of any for Values with an Unknown Type

Suppose you want to write a YAML parser (YAML can represent the same set of values as JSON but allows a superset of JSON's syntax). What should the return type of your `parseYAML` method be? It's tempting to make it `any` (like `JSON.parse`):

```
function parseYAML(yaml: string): any {
  // ...
}
```

But this flies in the face of Item 38's advice to avoid "contagious" `any` types, specifically by not returning them from functions.

Ideally you'd like your users to immediately assign the result to another type:

```
interface Book {
  name: string;
  author: string;
}
const book: Book = parseYAML(`
  name: Wuthering Heights
  author: Emily Brontë
`);
```

Without the type declarations, though, the book variable would quietly get an any type, thwarting type checking wherever it's used:

```
const book = parseYAML(`
  name: Jane Eyre
  author: Charlotte Brontë
`);
alert(book.title);  // No error, alerts "undefined" at runtime
book('read');  // No error, throws "TypeError: book is not a
               // function" at runtime
```

A safer alternative would be to have parseYAML return an unknown type:

```
function safeParseYAML(yaml: string): unknown {
  return parseYAML(yaml);
}
const book = safeParseYAML(`
  name: The Tenant of Wildfell Hall
  author: Anne Brontë
`);
alert(book.title);
   // ~~~~ Object is of type 'unknown'
book("read");
// ~~~~~~~~~~ Object is of type 'unknown'
```

To understand the unknown type, it helps to think about any in terms of assignability. The power and danger of any come from two properties:

- Any type is assignable to the any type.
- The any type is assignable to any other type.[2]

In the context of "thinking of types as sets of values" (Item 7), any clearly doesn't fit into the type system, since a set can't simultaneously be both a subset and a superset of all other sets. This is the source of any's power but also the reason it's problematic. Since the type checker is set-based, the use of any effectively disables it.

The unknown type is an alternative to any that *does* fit into the type system. It has the first property (any type is assignable to unknown) but not the second (unknown is only

2 With the exception of never.

assignable to unknown and, of course, any). The never type is the opposite: it has the second property (can be assigned to any other type) but not the first (nothing can be assigned to never).

Attempting to access a property on a value with the unknown type is an error. So is attempting to call it or do arithmetic with it. You can't do much with unknown, which is exactly the point. The errors about an unknown type will encourage you to add an appropriate type:

```
const book = safeParseYAML(`
  name: Villette
  author: Charlotte Brontë
`) as Book;
alert(book.title);
        // ~~~~~ Property 'title' does not exist on type 'Book'
book('read');
// ~~~~~~~~~ this expression is not callable
```

These errors are more sensible. Since unknown is not assignable to other types, a type assertion is required. But it is also appropriate: we really do know more about the type of the resulting object than TypeScript does.

unknown is appropriate whenever you know that there will be a value but you don't know its type. The result of parseYAML is one example, but there are others. In the GeoJSON spec, for example, the properties property of a Feature is a grab-bag of anything JSON serializable. So unknown makes sense:

```
interface Feature {
  id?: string | number;
  geometry: Geometry;
  properties: unknown;
}
```

A type assertion isn't the only way to recover a type from an unknown object. An instanceof check will do:

```
function processValue(val: unknown) {
  if (val instanceof Date) {
    val  // Type is Date
  }
}
```

You can also use a user-defined type guard:

```
function isBook(val: unknown): val is Book {
  return (
      typeof(val) === 'object' && val !== null &&
      'name' in val && 'author' in val
  );
}
function processValue(val: unknown) {
```

```
    if (isBook(val)) {
      val;  // Type is Book
    }
  }
```

TypeScript requires quite a bit of proof to narrow an `unknown` type: in order to avoid errors on the `in` checks, you first have to demonstrate that `val` is an object type and that it is non-`null` (since `typeof null === 'object'`).

You'll sometimes see a generic parameter used instead of `unknown`. You could have declared the `safeParseYAML` function this way:

```
function safeParseYAML<T>(yaml: string): T {
  return parseYAML(yaml);
}
```

This is generally considered bad style in TypeScript, however. It looks different than a type assertion, but is functionally the same. Better to just return `unknown` and force your users to use an assertion or narrow to the type they want.

`unknown` can also be used instead of `any` in "double assertions":

```
declare const foo: Foo;
let barAny = foo as any as Bar;
let barUnk = foo as unknown as Bar;
```

These are functionally equivalent, but the `unknown` form has less risk if you do a refactor and break up the two assertions. In that case the `any` could escape and spread. If the `unknown` type escapes, it will probably just produce an error.

As a final note, you may see code that uses `object` or `{}` in a similar way to how `unknown` has been described in this item. They are also broad types but are slightly narrower than `unknown`:

- The `{}` type consists of all values except `null` and `undefined`.
- The `object` type consists of all non-primitive types. This doesn't include `true` or `12` or `"foo"` but does include objects and arrays.

The use of `{}` was more common before the `unknown` type was introduced. Uses today are somewhat rare: only use `{}` instead of `unknown` if you really do know that `null` and `undefined` aren't possibilities.

Things to Remember

- The `unknown` type is a type-safe alternative to `any`. Use it when you know you have a value but do not know what its type is.
- Use `unknown` to force your users to use a type assertion or do type checking.

- Understand the difference between `{}`, `object`, and `unknown`.

Item 43: Prefer Type-Safe Approaches to Monkey Patching

One of the most famous features of JavaScript is that its objects and classes are "open" in the sense that you can add arbitrary properties to them. This is occasionally used to create global variables on web pages by assigning to `window` or `document`:

```
window.monkey = 'Tamarin';
document.monkey = 'Howler';
```

or to attach data to DOM elements:

```
const el = document.getElementById('colobus');
el.home = 'tree';
```

This style is particularly common with code that uses jQuery.

You can even attach properties to the prototypes of built-ins, with sometimes surprising results:

```
>  RegExp.prototype.monkey = 'Capuchin'
"Capuchin"
> /123/.monkey
"Capuchin"
```

These approaches are generally not good designs. When you attach data to `window` or a DOM node, you are essentially turning it into a global variable. This makes it easy to inadvertently introduce dependencies between far-flung parts of your program and means that you have to think about side effects whenever you call a function.

Adding TypeScript introduces another problem: while the type checker knows about built-in properties of `Document` and `HTMLElement`, it certainly doesn't know about the ones you've added:

```
document.monkey = 'Tamarin';
      // ~~~~~~ Property 'monkey' does not exist on type 'Document'
```

The most straightforward way to fix this error is with an `any` assertion:

```
(document as any).monkey = 'Tamarin';  // OK
```

This satisfies the type checker, but, as should be no surprise by now, it has some downsides. As with any use of `any`, you lose type safety and language services:

```
(document as any).monky = 'Tamarin';  // Also OK, misspelled
(document as any).monkey = /Tamarin/;  // Also OK, wrong type
```

The best solution is to move your data out of `document` or the DOM. But if you can't (perhaps you're using a library that requires it or are in the process of migrating a JavaScript application), then you have a few next-best options available.

One is to use an augmentation, one of the special abilities of `interface` (Item 13):

```
interface Document {
  /** Genus or species of monkey patch */
  monkey: string;
}

document.monkey = 'Tamarin';  // OK
```

This is an improvement over using `any` in a few ways:

- You get type safety. The type checker will flag misspellings or assignments of the wrong type.
- You can attach documentation to the property (Item 48).
- You get autocomplete on the property.
- There is a record of precisely what the monkey patch is.

In a module context (i.e., a TypeScript file that uses `import` / `export`), you'll need to add a `declare global` to make this work:

```
export {};
declare global {
  interface Document {
    /** Genus or species of monkey patch */
    monkey: string;
  }
}
document.monkey = 'Tamarin';  // OK
```

The main issues with using an augmentation have to do with scope. First, the augmentation applies globally. You can't hide it from other parts of your code or from libraries. And second, if you assign the property while your application is running, there's no way to introduce the augmentation only after this has happened. This is particularly problematic when you patch HTML Elements, where some elements on the page will have the property and some will not. For this reason, you might want to declare the property to be `string|undefined`. This is more accurate, but will make the type less convenient to work with.

Another approach is to use a more precise type assertion:

```
interface MonkeyDocument extends Document {
  /** Genus or species of monkey patch */
  monkey: string;
}
```

```
(document as MonkeyDocument).monkey = 'Macaque';
```

TypeScript is OK with the type assertion because `Document` and `MonkeyDocument` share properties (Item 9). And you get type safety in the assignment. The scope issues are also more manageable: there's no global modification of the `Document` type, just the introduction of a new type (which is only in scope if you import it). You have to write an assertion (or introduce a new variable) whenever you reference the monkey-patched property. But you can take that as encouragement to refactor into something more structured. Monkey patching shouldn't be *too* easy!

Things to Remember

- Prefer structured code to storing data in globals or on the DOM.
- If you must store data on built-in types, use one of the type-safe approaches (augmentation or asserting a custom interface).
- Understand the scoping issues of augmentations.

Item 44: Track Your Type Coverage to Prevent Regressions in Type Safety

Are you safe from the problems associated with any types once you've added type annotations for values with implicit `any` types and enabled `noImplicitAny`? The answer is "no"; `any` types can still enter your program in two main ways:

Explicit `any` *types*
: Even if you follow the advice of Items 38 and 39, making your `any` types both narrow and specific, they remain `any` types. In particular, types like `any[]` and `{[key: string]: any}` become plain `any`s once you index into them, and the resulting `any` types can flow through your code.

From third-party type declarations
: This is particularly insidious since `any` types from an `@types` declaration file enter silently: even though you have `noImplicitAny` enabled and you never typed `any`, you still have `any` types flowing through your code.

Because of the negative effects `any` types can have on type safety and developer experience (Item 5), it's a good idea to keep track of the number of them in your codebase. There are many ways to do this, including the `type-coverage` package on npm:

```
$ npx type-coverage
9985 / 10117 98.69%
```

This means that, of the 10,117 symbols in this project, 9,985 (98.69%) had a type other than any or an alias to any. If a change inadvertently introduces an any type and it flows through your code, you'll see a corresponding drop in this percentage.

In some ways this percentage is a way of keeping score on how well you've followed the advice of the other items in this chapter. Using narrowly scoped any will reduce the number of symbols with any types, and so will using more specific forms like any[]. Tracking this numerically helps you make sure things only get better over time.

Even collecting type coverage information once can be informative. Running type-coverage with the --detail flag will print where every any type occurs in your code:

```
$ npx type-coverage --detail
path/to/code.ts:1:10 getColumnInfo
path/to/module.ts:7:1 pt2
...
```

These are worth investigating because they're likely to turn up sources of anys that you hadn't considered. Let's look at a few examples.

Explicit any types are often the result of choices you made for expediency earlier on. Perhaps you were getting a type error that you didn't want to take the time to sort out. Or maybe the type was one that you hadn't written out yet. Or you might have just been in a rush.

Type assertions with any can prevent types from flowing where they otherwise would. Perhaps you've built an application that works with tabular data and needed a single-parameter function that built up some kind of column description:

```
function getColumnInfo(name: string): any {
  return utils.buildColumnInfo(appState.dataSchema, name);  // Returns any
}
```

The utils.buildColumnInfo function returned any at some point. As a reminder, you added a comment and an explicit ": any" annotation to the function.

However, in the intervening months you've also added a type for ColumnInfo, and utils.buildColumnInfo no longer returns any. The any annotation is now throwing away valuable type information. Get rid of it!

Third-party any types can come in a few forms, but the most extreme is when you give an entire module an any type:

```
declare module 'my-module';
```

Now you can import anything from my-module without error. These symbols all have any types and will lead to more any types if you pass values through them:

```
import {someMethod, someSymbol} from 'my-module';  // OK

const pt1 = {
  x: 1,
  y: 2,
};  // type is {x: number, y: number}
const pt2 = someMethod(pt1, someSymbol);  // OK, pt2's type is any
```

Since the usage looks identical to a well-typed module, it's easy to forget that you stubbed out the module. Or maybe a coworker did it and you never knew in the first place. It's worth revisiting these from time to time. Maybe there are official type declarations for the module. Or perhaps you've gained enough understanding of the module to write types yourself and contribute them back to the community.

Another common source of `any`s with third-party declarations is when there's a bug in the types. Maybe the declarations didn't follow the advice of Item 29 and declared a function to return a union type when in fact it returns something much more specific. When you first used the function this didn't seem worth fixing so you used an `any` assertion. But maybe the declarations have been fixed since then. Or maybe it's time to fix them yourself!

The considerations that led you to use an `any` type might not apply any more. Maybe there's a type you can plug in now where previously you used `any`. Maybe an unsafe type assertion is no longer necessary. Maybe the bug in the type declarations you were working around has been fixed. Tracking your type coverage highlights these choices and encourages you to keep revisiting them.

Things to Remember

- Even with `noImplicitAny` set, `any` types can make their way into your code either through explicit `any`s or third-party type declarations (`@types`).
- Consider tracking how well-typed your program is. This will encourage you to revisit decisions about using `any` and increase type safety over time.

CHAPTER 6

Types Declarations and @types

Dependency management can be confusing in any language, and TypeScript is no exception. This chapter will help you build a mental model for how dependencies work in TypeScript and show you how to work through some of the issues that can come up with them. It will also help you craft your own type declaration files to publish and share with others. By writing great type declarations, you can help not just your own project but the entire TypeScript community.

Item 45: Put TypeScript and @types in devDependencies

The Node Package Manager, npm, is ubiquitous in the JavaScript world. It provides both a repository of JavaScript libraries (the npm registry) and a way to specify which versions of them you depend on (*package.json*).

npm draws a distinction between a few types of dependencies, each of which goes in a separate section of *package.json*:

`dependencies`
: These are packages that are required to run your JavaScript. If you import `lodash` at runtime, then it should go in `dependencies`. When you publish your code on npm and another user installs it, it will also install these dependencies. (These are known as transitive dependencies.)

`devDependencies`
: These packages are used to develop and test your code but are not required at runtime. Your test framework would be an example of a `devDependency`. Unlike `dependencies`, these are *not* installed transitively with your packages.

`peerDependencies`
: These are packages that you require at runtime but don't want to be responsible for tracking. The canonical example is a plug-in. Your jQuery plug-in is compatible with a range of versions of jQuery itself, but you'd prefer that the user select one, rather than you choosing for them.

Of these, `dependencies` and `devDependencies` are by far the most common. As you use TypeScript, be aware of which type of dependency you're adding. Because TypeScript is a development tool and TypeScript types do not exist at runtime (Item 3), packages related to TypeScript generally belong in `devDependencies`.

The first dependency to consider is TypeScript itself. It is possible to install TypeScript system-wide, but this is generally a bad idea for two reasons:

- There's no guarantee that you and your coworkers will always have the same version installed.
- It adds a step to your project setup.

Make TypeScript a `devDependency` instead. That way you and your coworkers will always get the correct version when you run `npm install`. And updating your TypeScript version follows the same pattern as updating any other package.

Your IDE and build tools will happily discover a version of TypeScript installed in this way. On the command line you can use `npx` to run the version of `tsc` installed by npm:

```
$ npx tsc
```

The next type of dependency to consider is *type dependencies* or `@types`. If a library itself does not come with TypeScript type declarations, then you may still be able to find typings on DefinitelyTyped, a community-maintained collection of type definitions for JavaScript libraries. Type definitions from DefinitelyTyped are published on the npm registry under the `@types` scope: `@types/jquery` has type definitions for the jQuery, `@types/lodash` has types for Lodash, and so on. These `@types` packages only contain the *types*. They don't contain the implementation.

Your `@types` dependencies should also be `devDependencies`, even if the package itself is a direct dependency. For example, to depend on React and its type declarations, you might run:

```
$ npm install react
$ npm install --save-dev @types/react
```

This will result in a *package.json* file that looks something like this:

```
{
  "devDependencies": {
    "@types/lodash": "^16.8.19",
    "typescript": "^3.5.3"
  },
  "dependencies": {
    "react": "^16.8.6"
  }
}
```

The idea here is that you should publish JavaScript, not TypeScript, and your JavaScript does not depend on the `@types` when you run it. There are a few things that can go wrong with `@types` dependencies, and the next item will delve deeper into this topic.

Things to Remember

- Avoid installing TypeScript system-wide. Make TypeScript a `devDependency` of your project to ensure that everyone on the team is using a consistent version.
- Put `@types` dependencies in `devDependencies`, not `dependencies`. If you need `@types` at runtime, then you may want to rework your process.

Item 46: Understand the Three Versions Involved in Type Declarations

Dependency management rarely conjures up happy feelings for software developers. Usually you just want to use a library and not think too much about whether its transitive dependencies are compatible with yours.

The bad news is that TypeScript doesn't make this any better. In fact, it makes dependency management quite a bit *more* complicated. This is because instead of having a single version to worry about, you now have three:

- The version of the package
- The version of its type declarations (`@types`)
- The version of TypeScript

If any of these versions get out of sync with one another, you can run into errors that may not be clearly related to dependency management. But as the saying goes, "make things as simple as possible but no simpler." Understanding the full complexity of TypeScript package management will help you diagnose and fix problems. And it will help you make more informed decisions when it comes time to publish type declarations of your own.

Here's how dependencies in TypeScript are supposed to work. You install a package as a direct dependency, and you install its types as a dev dependency (see Item 45):

```
$ npm install react
+ react@16.8.6

$ npm install --save-dev @types/react
+ @types/react@16.8.19
```

Note that the major and minor versions (`16.8`) match but that the patch versions (`.6` and `.19`) do not. This is exactly what you want to see. The `16.8` in the `@types` version means that these type declarations describe the API of version `16.8` of `react`. Assuming the `react` module follows good semantic versioning hygiene, the patch versions (`16.8.1`, `16.8.2`, ...) will not change its public API and will not require updates to the type declarations. But the type declarations *themselves* might have bugs or omissions. The patch versions of the `@types` module correspond to these sorts of fixes and additions. In this case, there were many more updates to the type declarations than the library itself (19 versus 6).

This can go wrong in a few ways.

First, you might update a library but forget to update its type declarations. In this case you'll get type errors whenever you try to use new features of the library. If there were breaking changes to the library, you might get runtime errors despite your code passing the type checker.

The solution is usually to update your type declarations so that the versions are back in sync. If the type declarations have not been updated, you have a few options. You can use an augmentation in your own project to add new functions and methods that you'd like to use. Or you can contribute updated type declarations back to the community.

Second, your type declarations might get ahead of your library. This can happen if you've been using a library without its typings (perhaps you gave it an `any` type using `declare module`) and try to install them later. If there have been new releases of the library and its type declarations, your versions might be out of sync. The symptoms of this are similar to the first problem, just in reverse. The type checker will be comparing your code against the latest API, while you'll be using an older one at runtime. The solution is to either upgrade the library or downgrade the type declarations until they match.

Third, the type declarations might require a newer version of TypeScript than you're using in your project. Much of the development of TypeScript's type system has been motivated by an attempt to more precisely type popular JavaScript libraries like Lodash, React, and Ramda. It makes sense that the type declarations for these libraries would want to use the latest and greatest features to get you better type safety.

If this happens, you'll experience it as type errors in the `@types` declarations themselves. The solution is to either upgrade your TypeScript version, use an older version of the type declarations, or, if you really can't update TypeScript, stub out the types with `declare module`. It is possible for a library to provide different type declarations for different versions of TypeScript via `typesVersions`, but this is rare: at the time of this writing, fewer than 1% of the packages on DefinitelyTyped did so.

To install `@types` for a specific version of TypeScript, you can use:

```
npm install --save-dev @types/lodash@ts3.1
```

The version matching between libraries and their types is best effort and may not always be correct. But the more popular the library is, the more likely it is that its type declarations will get this right.

Fourth, you can wind up with duplicate `@types` dependencies. Say you depend on `@types/foo` and `@types/bar`. If `@types/bar` depends on an incompatible version of `@types/foo`, then npm will attempt to resolve this by installing both versions, one in a nested folder:

```
node_modules/
  @types/
    foo/
      index.d.ts @1.2.3
    bar/
      index.d.ts
      node_modules/
        @types/
          foo/
            index.d.ts @2.3.4
```

While this is sometimes OK for node modules that are used at runtime, it almost certainly won't be OK for type declarations, which live in a flat global namespace. You'll see this as errors about duplicate declarations or declarations that cannot be merged. You can track down why you have a duplicate type declaration by running `npm ls @types/foo`. The solution is typically to update your dependency on `@types/foo` or `@types/bar` so that they are compatible. Transitive `@types` dependencies like these are often a source of trouble. If you're publishing types, see Item 51 for ways to avoid them.

Some packages, particularly those written in TypeScript, choose to bundle their own type declarations. This is usually indicated by a `"types"` field in their *package.json* which points to a *.d.ts* file:

```
{
  "name": "left-pad",
  "version": "1.3.0",
  "description": "String left pad",
  "main": "index.js",
  "types": "index.d.ts",
  // ...
}
```

Does this solve all our problems? Would I even be asking if the answer was "yes"?

Bundling types *does* solve the problem of version mismatch, particularly if the library itself is written in TypeScript and the type declarations are generated by `tsc`. But bundling has some problems of its own.

First, what if there's an error in the bundled types that can't be fixed through augmentation? Or the types worked fine when they were published, but a new TypeScript version has since been released which flags an error. With `@types` you could depend on the library's implementation but not its type declarations. But with bundled types, you lose this option. One bad type declaration might keep you stuck on an old version of TypeScript. Contrast this with DefinitelyTyped: as TypeScript is developed, Microsoft runs it against all the type declarations on DefinitelyTyped. Breaks are fixed quickly.

Second, what if your types depend on another library's type declarations? Usually this would be a `devDependency` (Item 45). But if you publish your module and another user installs it, they won't get your `devDependencies`. Type errors will result. On the other hand, you probably don't want to make it a direct dependency either, since then your JavaScript users will install `@types` modules for no reason. Item 51 discusses the standard workaround for this situation. But if you publish your types on DefinitelyTyped, this is not a problem at all: you declare your type dependency there and only your TypeScript users will get it.

Third, what if you need to fix an issue with the type declarations of an old version of your library? Would you be able to go back and release a patch update? DefinitelyTyped has mechanisms for simultaneously maintaining type declarations for different versions of the same library, something that might be hard for you to do in your own project.

Fourth, how committed to accepting patches for type declarations are you? Remember the versions of `react` and `@types/react` from the start of this item. There were three times more patch updates to the type declarations than the library itself. DefinitelyTyped is community-maintained and is able to handle this volume. In particular, if a library maintainer doesn't look at a patch within five days, a global maintainer will. Can you commit to a similar turnaround time for your library?

Managing dependencies in TypeScript can be challenging, but it does come with rewards: well-written type declarations can help you learn how to use libraries correctly and can greatly improve your productivity with them. As you run into issues with dependency management, keep the three versions in mind.

If you are publishing packages, weigh the pros and cons of bundling type declarations versus publishing them on DefinitelyTyped. The official recommendation is to bundle type declarations only if the library is written in TypeScript. This works well in practice since `tsc` can automatically generate type declarations for you (using the `declaration` compiler option). For JavaScript libraries, handcrafted type declarations are more likely to contain errors, and they'll require more updates. If you publish your type declarations on DefinitelyTyped, the community will help you support and maintain them.

Things to Remember

- There are three versions involved in an `@types` dependency: the library version, the `@types` version, and the TypeScript version.
- If you update a library, make sure you update the corresponding `@types`.
- Understand the pros and cons of bundling types versus publishing them on DefinitelyTyped. Prefer bundling types if your library is written in TypeScript and DefinitelyTyped if it is not.

Item 47: Export All Types That Appear in Public APIs

Use TypeScript long enough and you'll eventually find yourself wanting to use a `type` or `interface` from a third-party module only to find that it isn't exported. Fortunately TypeScript's tools for mapping between types are rich enough that, as a library user, you can almost always find a way to reference the type you want. As a library author, this means that you ought to just export your types to begin with. If a type ever appears in a function declaration, it is effectively exported. So you may as well make things explicit.

Suppose you want to create some secret, unexported types:

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

interface SecretSanta {
  name: SecretName;
  gift: string;
}

export function getGift(name: SecretName, gift: string): SecretSanta {
  // ...
}
```

As a user of your module, I cannot directly import `SecretName` or `SecretSanta`, only `getGift`. But this is no barrier: because those types appear in an exported function signature, I can extract them. One way is to use the `Parameters` and `ReturnType` generic types:

```
type MySanta = ReturnType<typeof getGift>;  // SecretSanta
type MyName = Parameters<typeof getGift>[0];  // SecretName
```

If your goal in not exporting these types was to preserve flexibility, then the jig is up! You've already committed to them by putting them in a public API. Do your users a favor and export them.

Things to Remember

- Export types that appear in any form in any public method. Your users will be able to extract them anyway, so you may as well make it easy for them.

Item 48: Use TSDoc for API Comments

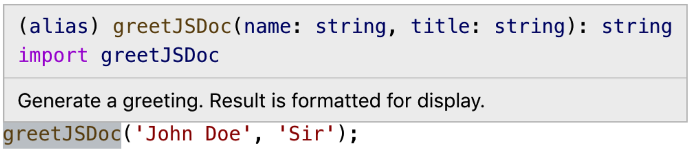
Here's a TypeScript function to generate a greeting:

```
// Generate a greeting. Result is formatted for display.
function greet(name: string, title: string) {
  return `Hello ${title} ${name}`;
}
```

The author was kind enough to leave a comment describing what this function does. But for documentation intended to be read by users of your functions, it's better to use JSDoc-style comments:

```
/** Generate a greeting. Result is formatted for display. */
function greetJSDoc(name: string, title: string) {
  return `Hello ${title} ${name}`;
}
```

The reason is that there is a nearly universal convention in editors to surface JSDoc-style comments when the function is called (see Figure 6-1).



```
(alias) greetJSDoc(name: string, title: string): string
import greetJSDoc
```

Generate a greeting. Result is formatted for display.

```
greetJSDoc('John Doe', 'Sir');
```

Figure 6-1. JSDoc-style comments are typically surfaced in tooltips in your editor.

Whereas the inline comment gets no such treatment (see Figure 6-2).

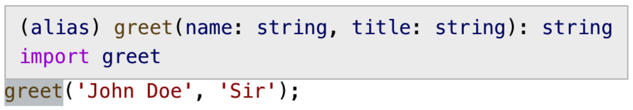

```
(alias) greet(name: string, title: string): string
import greet
greet('John Doe', 'Sir');
```

Figure 6-2. Inline comments are typically not shown in tooltips.

The TypeScript language service supports this convention, and you should take advantage of it. If a comment describes a public API, it should be JSDoc. In the context of TypeScript, these comments are sometimes called TSDoc. You can use many of the usual conventions like `@param` and `@returns`:

```
/**
 * Generate a greeting.
 * @param name Name of the person to greet
 * @param salutation The person's title
 * @returns A greeting formatted for human consumption.
 */
function greetFullTSDoc(name: string, title: string) {
  return `Hello ${title} ${name}`;
}
```

This lets editors show the relevant documentation for each parameter as you're writing out a function call (as shown in Figure 6-3).

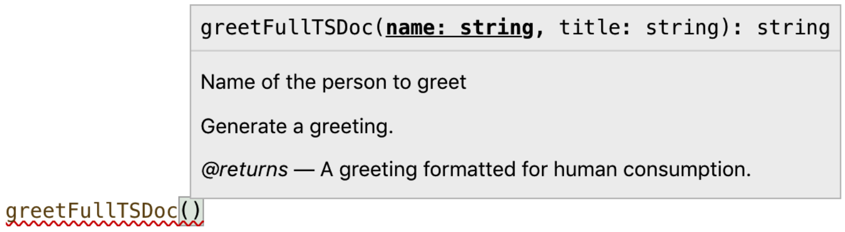
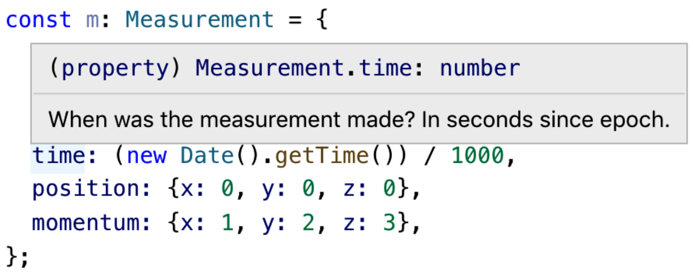

```
greetFullTSDoc(name: string, title: string): string

Name of the person to greet

Generate a greeting.

@returns — A greeting formatted for human consumption.
greetFullTSDoc()
```

Figure 6-3. An @param annotation lets your editor show documentation for the current parameter as you type it.

You can also use TSDoc with type definitions:

```
/** A measurement performed at a time and place. */
interface Measurement {
  /** Where was the measurement made? */
  position: Vector3D;
  /** When was the measurement made? In seconds since epoch. */
  time: number;
  /** Observed momentum */
  momentum: Vector3D;
}
```

As you inspect individual fields in a `Measurement` object, you'll get contextual documentation (see Figure 6-4).

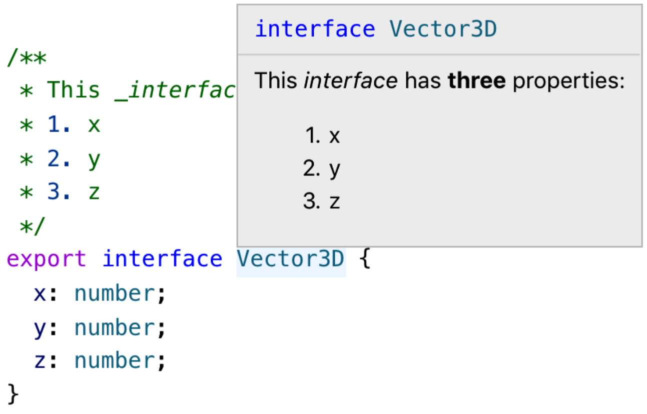


```
const m: Measurement = {
  (property) Measurement.time: number
  When was the measurement made? In seconds since epoch.
  time: (new Date().getTime()) / 1000,
  position: {x: 0, y: 0, z: 0},
  momentum: {x: 1, y: 2, z: 3},
};
```

Figure 6-4. TSDoc for a field is shown when you mouse over that field in your editor.

TSDoc comments are formatted using Markdown, so if you want to use bold, italic, or bulleted lists, you can (see Figure 6-5):

```
/**
 * This _interface_ has **three** properties:
 * 1. x
 * 2. y
 * 3. z
 */
interface Vector3D {
  x: number;
  y: number;
  z: number;
}
```



```
/**
 * This _interfac
 * 1. x
 * 2. y
 * 3. z
 */
export interface Vector3D {
  x: number;
  y: number;
  z: number;
}
```

interface Vector3D

This *interface* has **three** properties:

1. x
2. y
3. z

Figure 6-5. TSDoc comments

Try to avoid writing essays in your documentation, though: the best comments are short and to the point.

JSDoc includes some conventions for specifying type information (`@param {string} name ...`), but you should avoid these in favor of TypeScript types (Item 30).

Things to Remember

- Use JSDoc-/TSDoc-formatted comments to document exported functions, classes, and types. This helps editors surface information for your users when it's most relevant.
- Use `@param`, `@returns`, and Markdown for formatting.
- Avoid including type information in documentation (see Item 30).

Item 49: Provide a Type for this in Callbacks

JavaScript's `this` keyword is one of the most notoriously confusing parts of the language. Unlike variables declared with `let` or `const`, which are lexically scoped, `this` is dynamically scoped: its value depends not on the way in which it was *defined* but on the way in which it was *called*.

`this` is most often used in classes, where it typically references the current instance of an object:

```
class C {
  vals = [1, 2, 3];
  logSquares() {
    for (const val of this.vals) {
      console.log(val * val);
    }
  }
}

const c = new C();
c.logSquares();
```

This logs:

```
1
4
9
```

Now look what happens if you try to put `logSquares` in a variable and call that:

```
const c = new C();
const method = c.logSquares;
method();
```

This version throws an error at runtime:

```
Uncaught TypeError: Cannot read property 'vals' of undefined
```

The problem is that `c.logSquares()` actually does two things: it calls `C.prototype.logSquares` *and* it binds the value of `this` in that function to `c`. By pulling out a reference to `logSquares`, you've separated these, and `this` gets set to `undefined`.

JavaScript gives you complete control over `this` binding. You can use `call` to explicitly set `this` and fix the problem:

```
const c = new C();
const method = c.logSquares;
method.call(c);  // Logs the squares again
```

There's no reason that `this` had to be bound to an instance of `C`. It could have been bound to anything. So libraries can, and do, make the value of `this` part of their APIs. Even the DOM makes use of this. In an event handler, for instance:

```
document.querySelector('input')!.addEventListener('change', function(e) {
  console.log(this);  // Logs the input element on which the event fired.
});
```

`this` binding often comes up in the context of callbacks like this one. If you want to define an `onClick` handler in a class, for example, you might try this:

```
class ResetButton {
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick() {
    alert(`Reset ${this}`);
  }
}
```

When `Button` calls `onClick`, it will alert "Reset undefined." Oops! As usual, the culprit is `this` binding. A common solution is to create a bound version of the method in the constructor:

```
class ResetButton {
  constructor() {
    this.onClick = this.onClick.bind(this);
  }
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick() {
    alert(`Reset ${this}`);
  }
}
```

The `onClick() { ... }` definition defines a property on `ResetButton.prototype`. This is shared by all instances of `ResetButton`. When you bind `this.onClick = ...` in the constructor, it creates a property called `onClick` on the instance of `ResetButton` with `this` bound to that instance. The `onClick` instance property comes before the `onClick` prototype property in the lookup sequence, so `this.onClick` refers to the bound function in the `render()` method.

There is a shorthand for binding that can sometimes be convenient:

```
class ResetButton {
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick = () => {
    alert(`Reset ${this}`);  // "this" always refers to the ResetButton instance.
  }
}
```

Here we've replaced `onClick` with an arrow function. This will define a new function every time a `ResetButton` is constructed with `this` set to the appropriate value. It's instructive to look at the JavaScript that this generates:

```
class ResetButton {
  constructor() {
    var _this = this;
    this.onClick = function () {
      alert("Reset " + _this);
    };
  }
  render() {
    return makeButton({ text: 'Reset', onClick: this.onClick });
  }
}
```

So what does this all have to do with TypeScript? Because `this` binding is part of JavaScript, TypeScript models it. This means that if you're writing (or typing) a library that sets the value of `this` on callbacks, then you should model this, too.

You do this by adding a `this` parameter to your callback:

```
function addKeyListener(
  el: HTMLElement,
  fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
  el.addEventListener('keydown', e => {
    fn.call(el, e);
  });
}
```

The this parameter is special: it's not just another positional argument. You can see this if you try to call it with two parameters:

```
function addKeyListener(
  el: HTMLElement,
  fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
  el.addEventListener('keydown', e => {
    fn(el, e);
        // ~ Expected 1 arguments, but got 2
  });
}
```

Even better, TypeScript will enforce that you call the function with the correct this context:

```
function addKeyListener(
  el: HTMLElement,
  fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
  el.addEventListener('keydown', e => {
    fn(e);
 // ~~~~~ The 'this' context of type 'void' is not assignable
 //       to method's 'this' of type 'HTMLElement'
  });
}
```

As a user of this function, you can reference this in the callback and get full type safety:

```
declare let el: HTMLElement;
addKeyListener(el, function(e) {
  this.innerHTML;  // OK, "this" has type of HTMLElement
});
```

Of course, if you use an arrow function here, you'll override the value of this. TypeScript will catch the issue:

```
class Foo {
  registerHandler(el: HTMLElement) {
    addKeyListener(el, e => {
      this.innerHTML;
        // ~~~~~~~~~ Property 'innerHTML' does not exist on type 'Foo'
    });
  }
}
```

Don't forget about this! If you set the value of this in your callbacks, then it's part of your API, and you should include it in your type declarations.

Things to Remember

- Understand how `this` binding works.
- Provide a type for `this` in callbacks when it's part of your API.

Item 50: Prefer Conditional Types to Overloaded Declarations

How would you write a type declaration for this JavaScript function?

```
function double(x) {
  return x + x;
}
```

`double` can be passed either a `string` or a `number`. So you might use a union type:

```
function double(x: number|string): number|string;
function double(x: any) { return x + x; }
```

(These examples all make use of TypeScript's concept of function overloading. For a refresher, see Item 3.)

While this declaration is accurate, it's a bit imprecise:

```
const num = double(12);  // string | number
const str = double('x');  // string | number
```

When `double` is passed a `number`, it returns a `number`. And when it's passed a `string`, it returns a `string`. This declaration misses that nuance and will produce types that are hard to work with.

You might try to capture this relationship using a generic:

```
function double<T extends number|string>(x: T): T;
function double(x: any) { return x + x; }

const num = double(12);  // Type is 12
const str = double('x');  // Type is "x"
```

Unfortunately, in our zeal for precision we've overshot. The types are now a little *too* precise. When passed a `string` type, this `double` declaration will result in a `string` type, which is correct. But when passed a string *literal* type, the return type is the same string literal type. This is wrong: doubling `'x'` results in `'xx'`, not `'x'`.

Another option is to provide multiple type declarations. While TypeScript only allows you to write one implementation of a function, it allows you to write any number of type declarations. You can use this to improve the type of `double`:

```
function double(x: number): number;
function double(x: string): string;
function double(x: any) { return x + x; }

const num = double(12);  // Type is number
const str = double('x');  // Type is string
```

This is progress! But is this declaration correct? Unfortunately there's still a subtle bug. This type declaration will work with values that are either a `string` or a `number`, but not with values that could be either:

```
function f(x: number|string) {
  return double(x);
             // ~ Argument of type 'string | number' is not assignable
             //   to parameter of type 'string'
}
```

This call to `double` is safe and should return `string|number`. When you overload type declarations, TypeScript processes them one by one until it finds a match. The error you're seeing is a result of the last overload (the `string` version) failing, because `string|number` is not assignable to `string`.

While you could patch this issue by adding a third `string|number` overload, the best solution is to use a *conditional type*. Conditional types are like if statements (conditionals) in type space. They're perfect for situations like this one where there are a few possibilities that you need to cover:

```
function double<T extends number | string>(
  x: T
): T extends string ? string : number;
function double(x: any) { return x + x; }
```

This is similar to the first attempt to type `double` using a generic, but with a more elaborate return type. You read the conditional type like you'd read a ternary (`?:`) operator in JavaScript:

- If `T` is a subset of `string` (e.g., `string` or a string literal or a union of string literals), then the return type is `string`.
- Otherwise return `number`.

With this declaration, all of our examples work:

```
const num = double(12);  // number
const str = double('x');  // string

// function f(x: string | number): string | number
function f(x: number|string) {
  return double(x);
}
```

The `number|string` example works because conditional types distribute over unions. When `T` is `number|string`, TypeScript resolves the conditional type as follows:

```
   (number|string) extends string ? string : number
-> (number extends string ? string : number) |
   (string extends string ? string : number)
-> number | string
```

While the type declaration using overloading was simpler to write, the version using conditional types is more correct because it generalizes to the union of the individual cases. This is often the case for overloads. Whereas overloads are treated independently, the type checker can analyze conditional types as a single expression, distributing them over unions. If you find yourself writing an overloaded type declarations, consider whether it might be better expressed using a conditional type.

Things to Remember

- Prefer conditional types to overloaded type declarations. By distributing over unions, conditional types allow your declarations to support union types without additional overloads.

Item 51: Mirror Types to Sever Dependencies

Suppose you've written a library for parsing CSV files. Its API is simple: you pass in the contents of the CSV file and get back a list of objects mapping column names to values. As a convenience for your NodeJS users, you allow the contents to be either a `string` or a NodeJS `Buffer`:

```
function parseCSV(contents: string | Buffer): {[column: string]: string}[]  {
  if (typeof contents === 'object') {
    // It's a buffer
    return parseCSV(contents.toString('utf8'));
  }
  // ...
}
```

The type definition for `Buffer` comes from the NodeJS type declarations, which you must install:

```
npm install --save-dev @types/node
```

When you publish your CSV parsing library, you include the type declarations with it. Since your type declarations depend on the NodeJS types, you include these as a `devDependency` (Item 45). If you do this, you're liable to get complaints from two groups of users:

- JavaScript developers who wonder what these `@types` modules are that they're depending on.
- TypeScript web developers who wonder why they're depending on NodeJS.

These complaints are reasonable. The `Buffer` behavior isn't essential and is only relevant for users who are using NodeJS already. And the declaration in `@types/node` is only relevant to NodeJS users who are also using TypeScript.

TypeScript's structural typing (Item 4) can help you out of the jam. Rather than using the declaration of `Buffer` from `@types/node`, you can write your own with just the methods and properties you need. In this case that's just a `toString` method that accepts an encoding:

```
interface CsvBuffer {
  toString(encoding: string): string;
}
function parseCSV(contents: string | CsvBuffer): {[column: string]: string}[]  {
  // ...
}
```

This interface is dramatically shorter than the complete one, but it does capture our (simple) needs from a `Buffer`. In a NodeJS project, calling `parseCSV` with a real `Buffer` is still OK because the types are compatible:

```
parseCSV(new Buffer("column1,column2\nval1,val2", "utf-8"));  // OK
```

If your library only depends on the types for another library, rather than its implementation, consider mirroring just the declarations you need into your own code. This will result in a similar experience for your TypeScript users and an improved experience for everyone else.

If you depend on the implementation of a library, you may still be able to apply the same trick to avoid depending on its typings. But this becomes increasingly difficult as the dependence grows larger and more essential. If you're copying a large portion of the type declarations for another library, you may want to formalize the relationship by making the `@types` dependency explicit.

This technique is also helpful for severing dependencies between your unit tests and production systems. See the `getAuthors` example in Item 4.

Things to Remember

- Use structural typing to sever dependencies that are nonessential.
- Don't force JavaScript users to depend on `@types`. Don't force web developers to depend on NodeJS.

Item 52: Be Aware of the Pitfalls of Testing Types

You wouldn't publish code without writing tests for it (I hope!), and you shouldn't publish type declarations without writing tests for them, either. But how do you test types? If you're authoring type declarations, testing is an essential but surprisingly fraught undertaking. It's tempting to make assertions about types inside the type system using the tools that TypeScript provides. But there are several pitfalls with this approach. Ultimately it's safer and more straightforward to use `dtslint` or a similar tool that inspects types from outside of the type system.

Suppose you've written a type declaration for a `map` function provided by a utility library (the popular Lodash and Underscore libraries both provide such a function):

```
declare function map<U, V>(array: U[], fn: (u: U) => V): V[];
```

How can you check that this type declaration results in the expected types? (Presumably there are separate tests for the implementation.) One common technique is to write a test file that calls the function:

```
map(['2017', '2018', '2019'], v => Number(v));
```

This will do some blunt error checking: if your declaration of `map` only listed a single parameter, this would catch the mistake. But does it feel like something is missing here?

The equivalent of this style of test for runtime behavior might look something like this:

```
test('square a number', () => {
  square(1);
  square(2);
});
```

Sure, this tests that the `square` function doesn't throw an error. But it's missing any checks on the return value, so there's no real test of the behavior. An incorrect implementation of `square` would still pass this test.

This approach is common in testing type declaration files because it's simple to copy over existing unit tests for a library. And while it does provide some value, it would be much better to actually check some types!

One way is to assign the result to a variable with a specific type:

```
const lengths: number[] = map(['john', 'paul'], name => name.length);
```

This is exactly the sort of superfluous type declaration that Item 19 would encourage you to remove. But here it plays an essential role: it provides some confidence that the `map` declaration is at least doing something sensible with the types. And indeed you can find many type declarations in DefinitelyTyped that use exactly this approach for testing. But, as we'll see, there are a few fundamental problems with using assignment for testing.

One is that you have to create a named variable that is likely to be unused. This adds boilerplate, but also means that you'll have to disable some forms of linting.

A common workaround is to define a helper:

```
function assertType<T>(x: T) {}

assertType<number[]>(map(['john', 'paul'], name => name.length));
```

This eliminates the unused variable issue, but there are still surprises.

A second issue is that we're checking *assignability* of the two types rather than equality. Often this works as you'd expect. For example:

```
const n = 12;
assertType<number>(n);  // OK
```

If you inspect the `n` symbol, you'll see that its type is actually `12`, a numeric literal type. This is a subtype of `number` and so the assignability check passes, just as you'd expect.

So far so good. But things get murkier when you start checking the types of objects:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
assertType<{name: string}[]>(
  map(beatles, name => ({
    name,
    inYellowSubmarine: name === 'ringo'
  })));  // OK
```

The `map` call returns an array of `{name: string, inYellowSubmarine: boolean}` objects. This is assignable to `{name: string}[]`, sure, but shouldn't we be forced to acknowledge the yellow submarine? Depending on the context you may or may not really want to check for type equality.

If your function returns another function, you may be surprised at what's considered assignable:

```
const add = (a: number, b: number) => a + b;
assertType<(a: number, b: number) => number>(add);  // OK

const double = (x: number) => 2 * x;
assertType<(a: number, b: number) => number>(double);  // OK!?
```

Are you surprised that the second assertion succeeds? The reason is that a function in TypeScript is assignable to a function type, which takes fewer parameters:

```
const g: (x: string) => any = () => 12;  // OK
```

This reflects the fact that it's perfectly fine to call a JavaScript function with more parameters than it's declared to take. TypeScript chooses to model this behavior rather than bar it, largely because it is pervasive in callbacks. The callback in the Lodash `map` function, for example, takes up to three parameters:

```
map(array, (name, index, array) => { /* ... */ });
```

While all three are available, it's very common to use only one or sometimes two, as we have so far in this item. In fact, it's quite rare to use all three. By disallowing this assignment, TypeScript would report errors in an enormous amount of JavaScript code.

So what can you do? You could break apart the function type and test its pieces using the generic `Parameters` and `ReturnType` types:

```
const double = (x: number) => 2 * x;
let p: Parameters<typeof double> = null!;
assertType<[number, number]>(p);
//                             ~ Argument of type '[number]' is not
//                               assignable to parameter of type [number, number]
let r: ReturnType<typeof double> = null!;
assertType<number>(r);  // OK
```

But if "this" isn't complicated enough, there's another issue: `map` sets the value of `this` for its callback. TypeScript can model this behavior (see Item 49), so your type declaration should do so. And you should test it. How can we do that?

Our tests of `map` so far have been a bit black box in style: we've run an array and function through `map` and tested the type of the result, but we haven't tested the details of the intermediate steps. We can do so by filling out the callback function and verifying the types of its parameters and `this` directly:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
assertType<number[]>(map(
  beatles,
  function(name, i, array) {
// ~~~~~~~ Argument of type '(name: any, i: any, array: any) => any' is
//         not assignable to parameter of type '(u: string) => any'
    assertType<string>(name);
    assertType<number>(i);
    assertType<string[]>(array);
    assertType<string[]>(this);
                      // ~~~~ 'this' implicitly has type 'any'
    return name.length;
  }
));
```

This surfaced a few issues with our declaration of `map`. Note the use of a non-arrow function so that we could test the type of `this`.

Here is a declaration that passes the checks:

```
declare function map<U, V>(
  array: U[],
  fn: (this: U[], u: U, i: number, array: U[]) => V
): V[];
```

There remains a final issue, however, and it is a major one. Here's a complete type declaration file for our module that will pass even the most stringent tests for `map` but is worse than useless:

```
declare module 'overbar';
```

This assigns an `any` type to the *entire module*. Your tests will all pass, but you won't have any type safety. What's worse, every call to a function in this module will quietly produce an `any` type, contagiously destroying type safety throughout your code. Even with `noImplicitAny`, you can still get `any` types through type declarations.

Barring some advanced trickery, it's quite difficult to detect an `any` type from within the type system. This is why the preferred method for testing type declarations is to use a tool that operates *outside* the type checker.

For type declarations in the DefinitelyTyped repository, this tool is `dtslint`. It operates through specially formatted comments. Here's how you might write the last test for the `map` function using `dtslint`:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
map(beatles, function(
  name,  // $ExpectType string
  i,     // $ExpectType number
  array  // $ExpectType string[]
) {
  this   // $ExpectType string[]
  return name.length;
});  // $ExpectType number[]
```

Rather than checking assignability, `dtslint` inspects the type of each symbol and does a textual comparison. This matches how you'd manually test the type declarations in your editor: `dtslint` essentially automates this process. This approach does have some drawbacks: `number|string` and `string|number` are textually different but the same type. But so are `string` and `any`, despite being assignable to each other, which is really the point.

Testing type declarations is tricky business. You *should* test them. But be aware of the pitfalls of some of the common techniques and consider using a tool like `dtslint` to avoid them.

Things to Remember

- When testing types, be aware of the difference between equality and assignability, particularly for function types.

- For functions that use callbacks, test the inferred types of the callback parameters. Don't forget to test the type of `this` if it's part of your API.
- Be wary of `any` in tests involving types. Consider using a tool like `dtslint` for stricter, less error-prone checking.

CHAPTER 7

Writing and Running Your Code

This chapter is a bit of a grab bag: it covers some issues that come up in writing code (not types) as well as issues you may run into when you run your code.

Item 53: Prefer ECMAScript Features to TypeScript Features

The relationship between TypeScript and JavaScript has changed over time. When Microsoft first started work on TypeScript in 2010, the prevailing attitude around JavaScript was that it was a problematic language that needed to be fixed. It was common for frameworks and source-to-source compilers to add missing features like classes, decorators, and a module system to JavaScript. TypeScript was no different. Early versions included home-grown versions of classes, enums, and modules.

Over time TC39, the standards body that governs JavaScript, added many of these same features to the core JavaScript language. And the features they added were not compatible with the versions that existed in TypeScript. This left the TypeScript team in an awkward predicament: adopt the new features from the standard or break existing code?

TypeScript has largely chosen to do the latter and eventually articulated its current governing principle: TC39 defines the runtime while TypeScript innovates solely in the type space.

There are a few remaining features from before this decision. It's important to recognize and understand these, because they don't fit the pattern of the rest of the language. In general, I recommend avoiding them to keep the relationship between TypeScript and JavaScript as clear as possible.

Enums

Many languages model types that can take on a small set of values using *enumerations* or *enums*. TypeScript adds them to JavaScript:

```
enum Flavor {
  VANILLA = 0,
  CHOCOLATE = 1,
  STRAWBERRY = 2,
}

let flavor = Flavor.CHOCOLATE;  // Type is Flavor

Flavor  // Autocomplete shows: VANILLA, CHOCOLATE, STRAWBERRY
Flavor[0]  // Value is "VANILLA"
```

The argument for enums is that they provide more safety and transparency than bare numbers. But enums in TypeScript have some quirks. There are actually several variants on enums that all have subtly different behaviors:

- A number-valued enum (like `Flavor`). Any number is assignable to this, so it's not very safe. (It was designed this way to make bit flag structures possible.)
- A string-valued enum. This does offer type safety, and also more transparent values at runtime. But it's not structurally typed, unlike every other type in TypeScript (more on this momentarily).
- `const enum`. Unlike regular enums, const enums go away completely at runtime. If you changed to `const enum Flavor` in the previous example, the compiler would rewrite `Flavor.CHOCOLATE` as `0`. This also breaks our expectations around how the compiler behaves and still has the divergent behaviors between `string` and `number`-valued enums.
- `const enum` with the `preserveConstEnums` flag set. This emits runtime code for `const enum`s, just like for a regular `enum`.

That string-valued enums are nominally typed comes as a particular surprise, since every other type in TypeScript uses structural typing for assignability (see Item 4):

```
enum Flavor {
  VANILLA = 'vanilla',
  CHOCOLATE = 'chocolate',
  STRAWBERRY = 'strawberry',
}

let flavor = Flavor.CHOCOLATE;  // Type is Flavor
    flavor = 'strawberry';
 // ~~~~~~ Type '"strawberry"' is not assignable to type 'Flavor'
```

This has implications when you publish a library. Suppose you have a function that takes a `Flavor`:

```
function scoop(flavor: Flavor) { /* ... */ }
```

Because a `Flavor` at runtime is really just a string, it's fine for your JavaScript users to call it with one:

```
scoop('vanilla');  // OK in JavaScript
```

but your TypeScript users will need to import the `enum` and use that instead:

```
scoop('vanilla');
   // ~~~~~~~~~ '"vanilla"' is not assignable to parameter of type 'Flavor'

import {Flavor} from 'ice-cream';
scoop(Flavor.VANILLA);  // OK
```

These divergent experiences for JavaScript and TypeScript users are a reason to avoid string-valued enums.

TypeScript offers an alternative to enums that is less common in other languages: a union of literal types.

```
type Flavor = 'vanilla' | 'chocolate' | 'strawberry';

let flavor: Flavor = 'chocolate';  // OK
    flavor = 'mint chip';
 // ~~~~~~ Type '"mint chip"' is not assignable to type 'Flavor'
```

This offers as much safety as the enum and has the advantage of translating more directly to JavaScript. It also offers similarly strong autocomplete in your editor:

```
function scoop(flavor: Flavor) {
  if (flavor === 'v
                   // Autocomplete here suggests 'vanilla'
}
```

For more on this approach, see Item 33.

Parameter Properties

It's common to assign properties to a constructor parameter when initializing a class:

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

TypeScript provides a more compact syntax for this:

```
class Person {
  constructor(public name: string) {}
}
```

This is called a "parameter property," and it is equivalent to the code in the first example. There are a few issues to be aware of with parameter properties:

- They are one of the few constructs which generates code when you compile to JavaScript (`enum`s are another). Generally compilation just involves erasing types.
- Because the parameter is only used in generated code, the source looks like it has unused parameters.
- A mix of parameter and non-parameter properties can hide the design of your classes.

For example:

```
class Person {
  first: string;
  last: string;
  constructor(public name: string) {
    [this.first, this.last] = name.split(' ');
  }
}
```

This class has three properties (`first`, `last`, `name`), but this is hard to read off the code because only two are listed before the constructor. This gets worse if the constructor takes other parameters, too.

If your class consists *only* of parameter properties and no methods, you might consider making it an `interface` and using object literals. Remember that the two are assignable to one another because of structural typing Item 4:

```
class Person {
  constructor(public name: string) {}
}
const p: Person = {name: 'Jed Bartlet'};  // OK
```

Opinions are divided on parameter properties. While I generally avoid them, others appreciate the saved keystrokes. Be aware that they do not fit the pattern of the rest of TypeScript, and may in fact obscure that pattern for new developers. Try to avoid hiding the design of your class by using a mix of parameter and non-parameter properties.

Namespaces and Triple-Slash Imports

Before ECMAScript 2015, JavaScript didn't have an official module system. Different environments added this missing feature in different ways: Node.js used `require` and `module.exports` whereas AMD used a `define` function with a callback.

TypeScript also filled this gap with its own module system. This was done using a `module` keyword and "triple-slash" imports. After ECMAScript 2015 added an official module system, TypeScript added `namespace` as a synonym for `module`, to avoid confusion:

```
namespace foo {
  function bar() {}
}
```

```
/// <reference path="other.ts"/>
foo.bar();
```

Outside of type declarations, triple-slash imports and the `module` keyword are just a historical curiosity. In your own code, you should use ECMASCript 2015–style modules (`import` and `export`). See Item 58.

Decorators

Decorators can be used to annotate or modify classes, methods, and properties. For example, you could define a `logged` annotation that logs all calls to a method on a class:

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  @logged
  greet() {
    return "Hello, " + this.greeting;
  }
}

function logged(target: any, name: string, descriptor: PropertyDescriptor) {
  const fn = target[name];
  descriptor.value = function() {
    console.log(`Calling ${name}`);
    return fn.apply(this, arguments);
  };
}

console.log(new Greeter('Dave').greet());
// Logs:
// Calling greet
// Hello, Dave
```

This feature was initially added to support the Angular framework and requires the `experimentalDecorators` property to be set in `tsconfig.json`. Their implementation has not yet been standardized by TC39 at the time of this writing, so any code you write today using decorators is liable to break or become non-standard in the future. Unless you're using Angular or another framework that requires annotations and until they're standardized, don't use TypeScript's decorators.

Things to Remember

- By and large, you can convert TypeScript to JavaScript by removing all the types from your code.
- Enums, parameter properties, triple-slash imports, and decorators are historical exceptions to this rule.
- In order to keep TypeScript's role in your codebase as clear as possible, I recommend avoiding these features.

Item 54: Know How to Iterate Over Objects

This code runs fine, and yet TypeScript flags an error in it. Why?

```
const obj = {
  one: 'uno',
  two: 'dos',
  three: 'tres',
};
for (const k in obj) {
  const v = obj[k];
         // ~~~~~~ Element implicitly has an 'any' type
         //        because type ... has no index signature
}
```

Inspecting the `obj` and `k` symbols gives a clue:

```
const obj = { /* ... */ };
// const obj: {
//     one: string;
//     two: string;
//     three: string;
// }
for (const k in obj) {  // const k: string
  // ...
}
```

The type of `k` is `string`, but you're trying to index into an object whose type only has three specific keys: `'one'`, `'two'`, and `'three'`. There are strings other than these three, so this has to fail.

Plugging in a narrower type declaration for `k` fixes the issue:

```
let k: keyof typeof obj;  // Type is "one" | "two" | "three"
for (k in obj) {
```

```
    const v = obj[k];  // OK
  }
```

So the real question is: why is the type of `k` in the first example inferred as `string` rather than `"one" | "two" | "three"`?

To understand, let's look at a slightly different example involving an interface and a function:

```
interface ABC {
  a: string;
  b: string;
  c: number;
}

function foo(abc: ABC) {
  for (const k in abc) {  // const k: string
    const v = abc[k];
           // ~~~~~~ Element implicitly has an 'any' type
           //        because type 'ABC' has no index signature
  }
}
```

It's the same error as before. And you can "fix" it using the same sort of declaration (`let k: keyof ABC`). But in this case TypeScript is right to complain. Here's why:

```
const x = {a: 'a', b: 'b', c: 2, d: new Date()};
foo(x);  // OK
```

The function `foo` can be called with any value *assignable* to `ABC`, not just a value with "a," "b," and "c" properties. It's entirely possible that the value will have other properties, too (see Item 4). To allow for this, TypeScript gives `k` the only type it can be confident of, namely, `string`.

Using the `keyof` declaration would have another downside here:

```
function foo(abc: ABC) {
  let k: keyof ABC;
  for (k in abc) {  // let k: "a" | "b" | "c"
    const v = abc[k];  // Type is string | number
  }
}
```

If `"a" | "b" | "c"` is too narrow for `k`, then `string | number` is certainly too narrow for `v`. In the preceding example one of the values is a `Date`, but it could be anything. The types here give a false sense of certainty that could lead to chaos at runtime.

So what if you just want to iterate over the object's keys and values without type errors? `Object.entries` lets you iterate over both simultaneously:

```
function foo(abc: ABC) {
  for (const [k, v] of Object.entries(abc)) {
    k  // Type is string
    v  // Type is any
  }
}
```

While these types may be hard to work with, they are at least honest!

You should also be aware of the possibility of *prototype pollution*. Even in the case of an object literal that you define, for-in can produce additional keys:

```
> Object.prototype.z = 3; // Please don't do this!
> const obj = {x: 1, y: 2};
> for (const k in obj) { console.log(k); }
x
y
z
```

Hopefully this doesn't happen in a nonadversarial environment (you should never add enumerable properties to `Object.prototype`), but it is another reason that for-in produces `string` keys even for object literals.

If you want to iterate over the keys and values in an object, use either a `keyof` declaration (`let k: keyof T`) or `Object.entries`. The former is appropriate for constants or other situations where you know that the object won't have additional keys and you want precise types. The latter is more generally appropriate, though the key and value types are more difficult to work with.

Things to Remember

- Use `let k: keyof T` and a for-in loop to iterate objects when you know exactly what the keys will be. Be aware that any objects your function receives as parameters might have additional keys.
- Use `Object.entries` to iterate over the keys and values of any object.

Item 55: Understand the DOM hierarchy

Most of the items in this book are agnostic about where you run your TypeScript: in a web browser, on a server, on a phone. This one is different. If you're not working in a browser, skip ahead!

The DOM hierarchy is always present when you're running JavaScript in a web browser. When you use `document.getElementById` to get an element or `document.createElement` to create one, it's always a particular kind of element, even if you're not entirely familiar with the taxonomy. You call the methods and use the properties that you want and hope for the best.

With TypeScript, the hierarchy of DOM elements becomes more visible. Knowing your `Node`s from your `Element`s and `EventTarget`s will help you debug type errors and decide when type assertions are appropriate. Because so many APIs are based on the DOM, this is relevant even if you're using a framework like React or d3.

Suppose you want to track a user's mouse as they drag it across a `<div>`. You write some seemingly innocuous JavaScript:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
  const dragStart = [eDown.clientX, eDown.clientY];
  const handleUp = (eUp: Event) => {
    targetEl.classList.remove('dragging');
    targetEl.removeEventListener('mouseup', handleUp);
    const dragEnd = [eUp.clientX, eUp.clientY];
    console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
  }
  targetEl.addEventListener('mouseup', handleUp);
}
const div = document.getElementById('surface');
div.addEventListener('mousedown', handleDrag);
```

TypeScript's type checker flags no fewer than 11 errors in these 14 lines of code:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
// ~~~~~~~           Object is possibly 'null'.
//         ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
  const dragStart = [
     eDown.clientX, eDown.clientY];
        // ~~~~~~~                Property 'clientX' does not exist on 'Event'
        //               ~~~~~~~ Property 'clientY' does not exist on 'Event'
  const handleUp = (eUp: Event) => {
    targetEl.classList.remove('dragging');
//  ~~~~~~~~           Object is possibly 'null'.
//           ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
    targetEl.removeEventListener('mouseup', handleUp);
//  ~~~~~~~~ Object is possibly 'null'
    const dragEnd = [
       eUp.clientX, eUp.clientY];
        // ~~~~~~~                Property 'clientX' does not exist on 'Event'
        //             ~~~~~~~   Property 'clientY' does not exist on 'Event'
    console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
  }
  targetEl.addEventListener('mouseup', handleUp);
// ~~~~~~~ Object is possibly 'null'
}
```

```
    const div = document.getElementById('surface');
    div.addEventListener('mousedown', handleDrag);
// ~~~ Object is possibly 'null'
```

What went wrong? What's this `EventTarget`? And why might everything be `null`?

To understand the `EventTarget` errors it helps to dig into the DOM hierarchy a bit. Here's some HTML:

```
<p id="quote">and <i>yet</i> it moves</p>
```

If you open your browser's JavaScript console and get a reference to the `p` element, you'll see that it's an `HTMLParagraphElement`:

```
const p = document.getElementsByTagName('p')[0];
p instanceof HTMLParagraphElement
// True
```

An `HTMLParagraphElement` is a subtype of `HTMLElement`, which is a subtype of `Element`, which is a subtype of `Node`, which is a subtype of `EventTarget`. Here are some examples of types along the hierarchy:

Table 7-1. Types in the DOM Hierarchy

Type	Examples
EventTarget	`window`, `XMLHttpRequest`
Node	`document`, `Text`, `Comment`
Element	*includes HTMLElements, SVGElements*
HTMLElement	`<i>`, `<b>`
HTMLButtonElement	`<button>`

An `EventTarget` is the most generic of DOM types. All you can do with it is add event listeners, remove them, and dispatch events. With this in mind, the `classList` errors start to make a bit more sense:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
// ~~~~~~~           Object is possibly 'null'
//        ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
  // ...
}
```

As its name implies, an `Event`'s `currentTarget` property is an `EventTarget`. It could even be `null`. TypeScript has no reason to believe that it has a `classList` property. While an `EventTargets` *could* be an `HTMLElement` in practice, from the type system's perspective there's no reason it couldn't be `window` or `XMLHTTPRequest`.

Moving up the hierarchy we come to Node. A couple of examples of Nodes that are not Elements are text fragments and comments. For instance, in this HTML:

```
<p>
  And <i>yet</i> it moves
  <!-- quote from Galileo -->
</p>
```

the outermost element is an HTMLParagraphElement. As you can see here, it has children and childNodes:

```
> p.children
HTMLCollection [i]
> p.childNodes
NodeList(5) [text, i, text, comment, text]
```

children returns an HTMLCollection, an array-like structure containing just the child Elements (<i>yet</i>). childNodes returns a NodeList, an Array-like collection of Nodes. This includes not just Elements (<i>yet</i>) but also text fragments ("And," "it moves") and comments ("quote from Galileo").

What's the difference between an Element and an HTMLElement? There are non-HTML Elements including the whole hierarchy of SVG tags. These are SVGElements, which are another type of Element. What's the type of an <html> or <svg> tag? They're HTMLHtmlElement and SVGSvgElement.

Sometimes these specialized classes will have properties of their own—for example, an HTMLImageElement has a src property, and an HTMLInputElement has a value property. If you want to read one of these properties off a value, its type must be specific enough to have that property.

TypeScript's type declarations for the DOM make liberal use of literal types to try to get you the most specific type possible. For example:

```
document.getElementsByTagName('p')[0];  // HTMLParagraphElement
document.createElement('button');  // HTMLButtonElement
document.querySelector('div');  // HTMLDivElement
```

but this is not always possible, notably with document.getElementById:

```
document.getElementById('my-div');  // HTMLElement
```

While type assertions are generally frowned upon (Item 9), this is a case where you know more than TypeScript does and so they are appropriate. There's nothing wrong with this, so long as you know that #my-div is a div:

```
document.getElementById('my-div') as HTMLDivElement;
```

with strictNullChecks enabled, you will need to consider the case that document.getElementById returns null. Depending on whether this can really happen, you can either add an if statement or an assertion (!):

```
const div = document.getElementById('my-div')!;
```

These types are not specific to TypeScript. Rather, they are generated from the formal specification of the DOM. This is an example of the advice of Item 35 to generate types from specs when possible.

So much for the DOM hierarchy. What about the `clientX` and `clientY` errors?

```
function handleDrag(eDown: Event) {
  // ...
  const dragStart = [
     eDown.clientX, eDown.clientY];
        // ~~~~~~~                Property 'clientX' does not exist on 'Event'
        //                ~~~~~~~ Property 'clientY' does not exist on 'Event'
  // ...
}
```

In addition to the hierarchy for `Node`s and `Element`s, there is also a hierarchy for `Event`s. The Mozilla documentation currently lists no fewer than 52 types of `Event`!

Plain `Event` is the most generic type of event. More specific types include:

`UIEvent`
: Any sort of user interface event

`MouseEvent`
: An event triggered by the mouse such as a click

`TouchEvent`
: A touch event on a mobile device

`WheelEvent`
: An event triggered by rotating the scroll wheel

`KeyboardEvent`
: A key press

The problem in `handleDrag` is that the events are declared as `Event`, while `clientX` and `clientY` exist only on the more specific `MouseEvent` type.

So how can you fix the example from the start of this item? TypeScript's declarations for the DOM make extensive use of context (Item 26). Inlining the mousedown handler gives TypeScript more information to work with and removes most of the errors. You can also declare the parameter type to be `MouseEvent` rather than `Event`. Here's a version that uses both techniques to fix the errors:

```
function addDragHandler(el: HTMLElement) {
  el.addEventListener('mousedown', eDown => {
    const dragStart = [eDown.clientX, eDown.clientY];
    const handleUp = (eUp: MouseEvent) => {
      el.classList.remove('dragging');
```

```
      el.removeEventListener('mouseup', handleUp);
      const dragEnd = [eUp.clientX, eUp.clientY];
      console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
    }
    el.addEventListener('mouseup', handleUp);
  });
}

const div = document.getElementById('surface');
if (div) {
  addDragHandler(div);
}
```

The `if` statement at the end handles the possibility that there is no `#surface` element. If you know that this element exists, you could use an assertion instead (`div!`). `addDragHandler` requires a non-null `HTMLElement`, so this is an example of pushing `null` values to the perimeter (Item 31).

Things to Remember

- The DOM has a type hierarchy that you can usually ignore while writing JavaScript. But these types become more important in TypeScript. Understanding them will help you write TypeScript for the browser.
- Know the differences between `Node`, `Element`, `HTMLElement`, and `EventTarget`, as well as those between `Event` and `MouseEvent`.
- Either use a specific enough type for DOM elements and Events in your code or give TypeScript the context to infer it.

Item 56: Don't Rely on Private to Hide Information

JavaScript has historically lacked a way to make properties of a class private. The usual workaround is a convention of prefixing fields that are not part of a public API with underscores:

```
class Foo {
  _private = 'secret123';
}
```

But this only discourages users from accessing private data. It is easy to circumvent:

```
const f = new Foo();
f._private;  // 'secret123'
```

TypeScript adds `public`, `protected`, and `private` field modifiers that seem to provide some enforcement:

```
class Diary {
  private secret = 'cheated on my English test';
}

const diary = new Diary();
diary.secret
   // ~~~~~~ Property 'secret' is private and only
   //        accessible within class 'Diary'
```

But `private` is a feature of the type system and, like all features of the type system, it goes away at runtime (see Item 3). Here's what this snippet looks like when TypeScript compiles it to JavaScript (with `target=ES2017`):

```
class Diary {
  constructor() {
    this.secret = 'cheated on my English test';
  }
}
const diary = new Diary();
diary.secret;
```

The `private` indicator is gone, and your secret is out! Much like the `_private` convention, TypeScript's access modifiers only discourage you from accessing private data. With a type assertion, you can even access a private property from within TypeScript:

```
class Diary {
  private secret = 'cheated on my English test';
}

const diary = new Diary();
(diary as any).secret  // OK
```

In other words, *don't rely on* `private` *to hide information!*

So what should you do if you want something more robust? The traditional answer has been to take advantage of one of JavaScript's most reliable ways to hide information: closures. You can create one in a constructor:

```
declare function hash(text: string): number;

class PasswordChecker {
  checkPassword: (password: string) => boolean;
  constructor(passwordHash: number) {
    this.checkPassword = (password: string) => {
      return hash(password) === passwordHash;
    }
  }
}

const checker = new PasswordChecker(hash('s3cret'));
checker.checkPassword('s3cret');  // Returns true
```

JavaScript offers no way to access the passwordHash variable from outside of the constructor of PasswordChecker. This does have a few downsides, however: specifically, because passwordHash can't be seen outside the constructor, every method that uses it also has to be defined there. This results in a copy of each method being created for every class instance, which will lead to higher memory use. It also prevents other instances of the same class from accessing private data. Closures may be inconvenient, but they will certainly keep your data private!

A newer option is to use private fields, a proposed language feature that is solidifying as this book goes to print. In this proposal, to make a field private both for type checking and at runtime, prefix it with a #:

```
class PasswordChecker {
  #passwordHash: number;

  constructor(passwordHash: number) {
    this.#passwordHash = passwordHash;
  }

  checkPassword(password: string) {
    return hash(password) === this.#passwordHash;
  }
}

const checker = new PasswordChecker(hash('s3cret'));
checker.checkPassword('secret');  // Returns false
checker.checkPassword('s3cret');  // Returns true
```

The #passwordHash property is not accessible from outside the class. In contrast to the closure technique, it *is* accessible from class methods and from other instances of the same class. For ECMAScript targets that don't natively support private fields, a fallback implementation using WeakMaps is used instead. The upshot is that your data is still private. This proposal was stage 3 and support was being added to TypeScript as this book went to print. If you'd like to use it, check the TypeScript release notes to see if it's generally available.

Finally, if you are worried about *security*, rather than just encapsulation, then there are others concerns to be aware of such as modifications to built-in prototypes and functions.

Things to Remember

- The private access modifier is only enforced through the type system. It has no effect at runtime and can be bypassed with an assertion. Don't assume it will keep data hidden.
- For more reliable information hiding, use a closure.

Item 57: Use Source Maps to Debug TypeScript

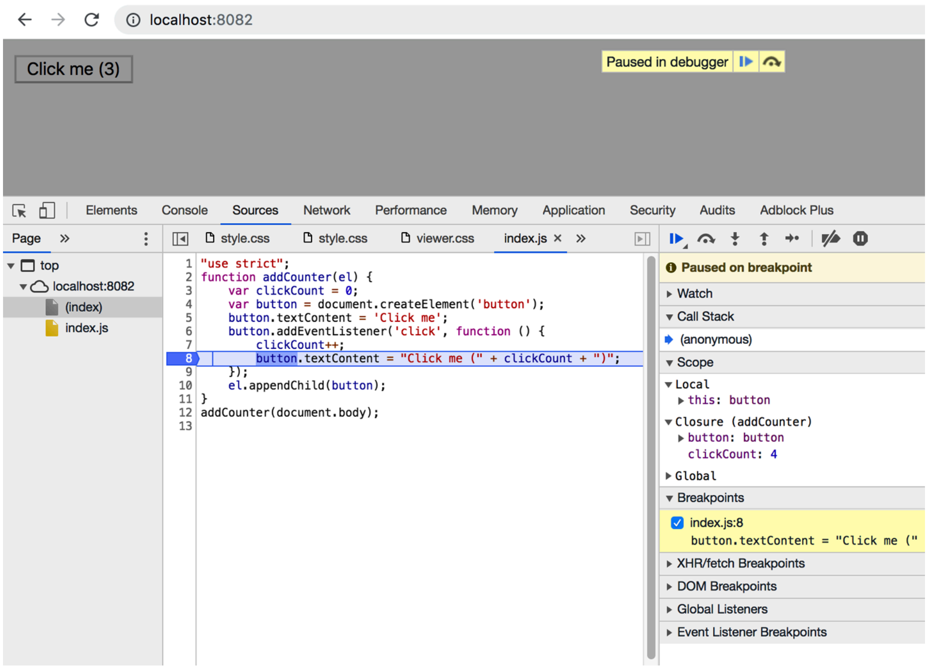
When you run TypeScript code, you're actually running the JavaScript that the TypeScript compiler generates. This is true of any source-to-source compiler, be it a minifier, a compiler, or a preprocessor. The hope is that this is mostly transparent, that you can pretend that the TypeScript source code is being executed without ever having to look at the JavaScript.

This works well until you have to debug your code. Debuggers generally work on the code you're executing and don't know about the translation process it went through. Since JavaScript is such a popular target language, browser vendors collaborated to solve this problem. The result is source maps. They map positions and symbols in a generated file back to the corresponding positions and symbols in the original source. Most browsers and many IDEs support them. If you're not using them to debug your TypeScript, you're missing out!

Suppose you've created a small script to add a button to an HTML page that increments every time you click it:

```
function addCounter(el: HTMLElement) {
  let clickCount = 0;
  const button = document.createElement('button');
  button.textContent = 'Click me';
  button.addEventListener('click', () => {
    clickCount++;
    button.textContent = `Click me (${clickCount})`;
  });
  el.appendChild(button);
}

addCounter(document.body);
```

If you load this in your browser and open the debugger, you'll see the generated JavaScript. This closely matches the original source, so debugging isn't too difficult, as you can see in Figure 7-1.



```
"use strict";
function addCounter(el) {
    var clickCount = 0;
    var button = document.createElement('button');
    button.textContent = 'Click me';
    button.addEventListener('click', function () {
        clickCount++;
        button.textContent = "Click me (" + clickCount + ")";
    });
    el.appendChild(button);
}
addCounter(document.body);
```

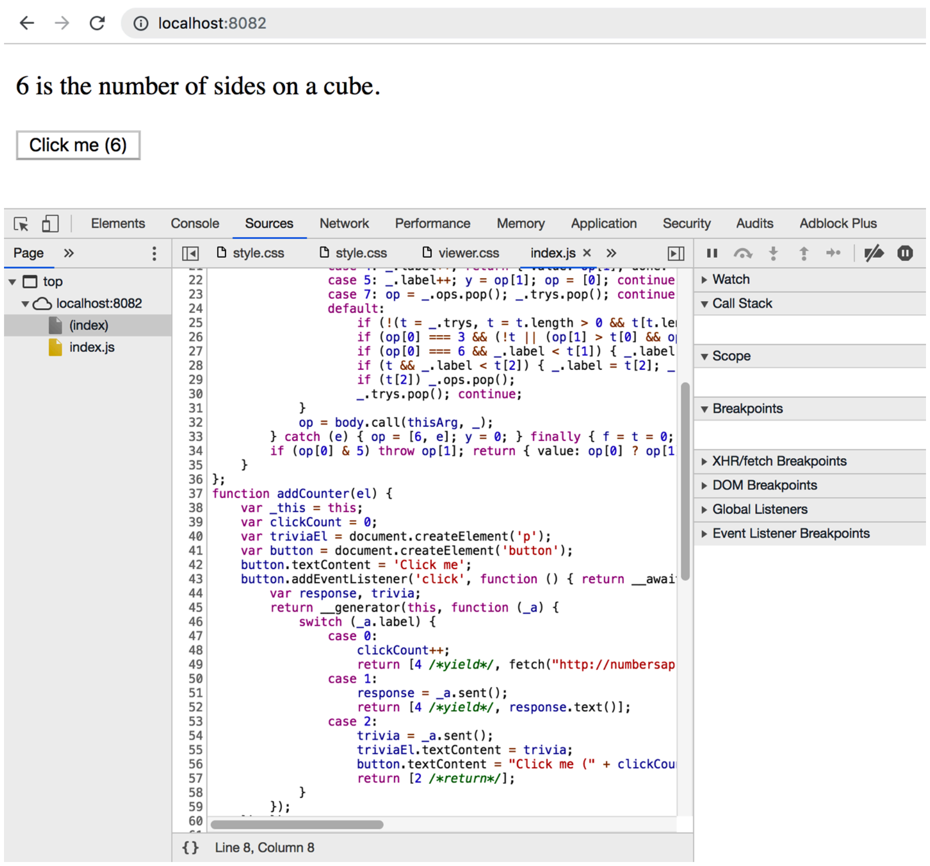
```
Paused on breakpoint
Watch
Call Stack
  (anonymous)
Scope
  Local
    this: button
  Closure (addCounter)
    button: button
    clickCount: 4
  Global
Breakpoints
  ✓ index.js:8
    button.textContent = "Click me ("
XHR/fetch Breakpoints
DOM Breakpoints
Global Listeners
Event Listener Breakpoints
```

Figure 7-1. Debugging generated JavaScript using Chrome's developer tools. For this simple example, the generated JavaScript closely resembles the TypeScript source.

Let's make the page more fun by fetching an interesting fact about each number from numbersapi.com:

```
function addCounter(el: HTMLElement) {
  let clickCount = 0;
  const triviaEl = document.createElement('p');
  const button = document.createElement('button');
  button.textContent = 'Click me';
  button.addEventListener('click', async () => {
    clickCount++;
    const response = await fetch(`http://numbersapi.com/${clickCount}`);
    const trivia = await response.text();
    triviaEl.textContent = trivia;
    button.textContent = `Click me (${clickCount})`;
  });
  el.appendChild(triviaEl);
  el.appendChild(button);
}
```

If you open up your browser's debugger now, you'll see that the generated source has gotten dramatically more complicated (see Figure 7-2).



6 is the number of sides on a cube.

```
                case 5: _.label++; y = op[1]; op = [0]; continue
                case 7: op = _.ops.pop(); _.trys.pop(); continue
                default:
                    if (!(t = _.trys, t = t.length > 0 && t[t.le
                    if (op[0] === 3 && (!t || (op[1] > t[0] && o
                    if (op[0] === 6 && _.label < t[1]) { _.label
                    if (t && _.label < t[2]) { _.label = t[2]; _
                    if (t[2]) _.ops.pop();
                    _.trys.pop(); continue;
            }
            op = body.call(thisArg, _);
        } catch (e) { op = [6, e]; y = 0; } finally { f = t = 0;
        if (op[0] & 5) throw op[1]; return { value: op[0] ? op[1
    }
};
function addCounter(el) {
    var _this = this;
    var clickCount = 0;
    var triviaEl = document.createElement('p');
    var button = document.createElement('button');
    button.textContent = 'Click me';
    button.addEventListener('click', function () { return __awai
        var response, trivia;
        return __generator(this, function (_a) {
            switch (_a.label) {
                case 0:
                    clickCount++;
                    return [4 /*yield*/, fetch("http://numbersap
                case 1:
                    response = _a.sent();
                    return [4 /*yield*/, response.text()];
                case 2:
                    trivia = _a.sent();
                    triviaEl.textContent = trivia;
                    button.textContent = "Click me (" + clickCou
                    return [2 /*return*/];
            }
        });
```

Figure 7-2. In this case the TypeScript compiler has generated JavaScript that doesn't closely resemble the original TypeScript source. This will make debugging more difficult.

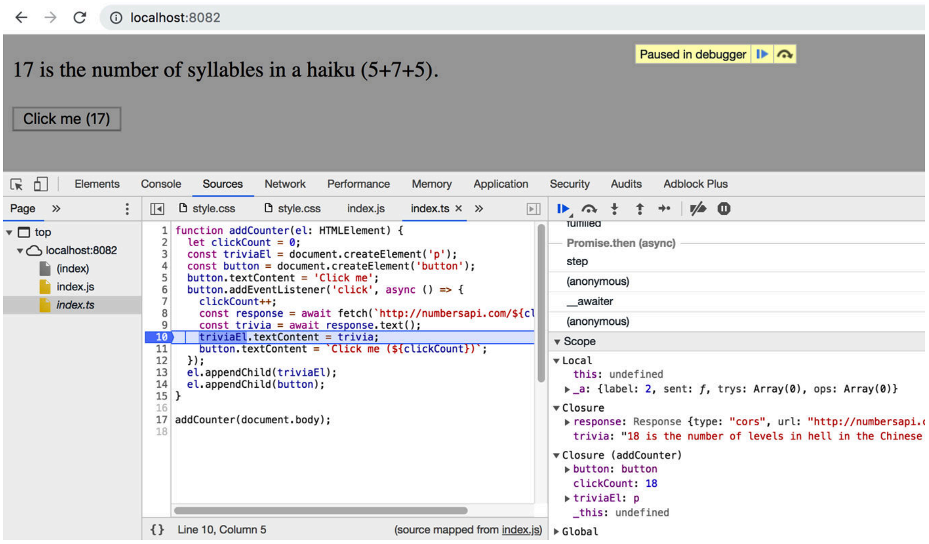
To support `async` and `await` in older browsers, TypeScript has rewritten the event handler as a state machine. This has the same behavior, but the code no longer bears such a close resemblance to the original source.

This is where source maps can help. To tell TypeScript to generate one, set the `source Map` option in your *tsconfig.json*:

```
{
  "compilerOptions": {
    "sourceMap": true
  }
}
```

Now when you run `tsc`, it generates two output files for each *.ts* file: a *.js* file and a *.js.map* file. The latter is the source map.

With this file in place, a new *index.ts* file appears in your browser's debugger. You can set breakpoints and inspect variables in it, just as you'd hope (see Figure 7-3).



```
function addCounter(el: HTMLElement) {
  let clickCount = 0;
  const triviaEl = document.createElement('p');
  const button = document.createElement('button');
  button.textContent = 'Click me';
  button.addEventListener('click', async () => {
    clickCount++;
    const response = await fetch(`http://numbersapi.com/${cl
    const trivia = await response.text();
    triviaEl.textContent = trivia;
    button.textContent = `Click me (${clickCount})`;
  });
  el.appendChild(triviaEl);
  el.appendChild(button);
}

addCounter(document.body);
```

```
▼Local
   this: undefined
  ▶_a: {label: 2, sent: ƒ, trys: Array(0), ops: Array(0)}
▼Closure
  ▶response: Response {type: "cors", url: "http://numbersapi.
   trivia: "18 is the number of levels in hell in the Chinese
▼Closure (addCounter)
  ▶button: button
   clickCount: 18
  ▶triviaEl: p
   _this: undefined
▶Global
```

Figure 7-3. When a source map is present, you can work with the original TypeScript source in your debugger, rather than the generated JavaScript.

Note that *index.ts* appears in italics in the file list on the left. This indicates that it isn't a "real" file in the sense that the web page included it. Rather, it was included via the source map. Depending on your settings, *index.js.map* will contain either a reference to *index.ts* (in which case the browser loads it over the network) or an inline copy of it (in which case no request is needed).

There are a few things to be aware of with source maps:

- If you are using a bundler or minifier with TypeScript, it may generate a source map of its own. To get the best debugging experience, you want this to map all the way back to the original TypeScript sources, not the generated JavaScript. If your bundler has built-in support for TypeScript, then this should just work. If not, you may need to hunt down some flags to make it read source map inputs.
- Be aware of whether you're serving source maps in production. The browser won't load source maps unless the debugger is open, so there's no performance impact for end users. But if the source map contains an inline copy of your original source code, then there may be content that you didn't intend to publicize. Does the world really need to see your snarky comments or internal bug tracker URLs?

You can also debug NodeJS programs using source maps. This is typically done via your editor or by connecting to your node process from a browser's debugger. Consult the Node docs for details.

The type checker can catch many errors before you run your code, but it is no substitute for a good debugger. Use source maps to get a great TypeScript debugging experience.

Things to Remember

- Don't debug generated JavaScript. Use source maps to debug your TypeScript code at runtime.
- Make sure that your source maps are mapped all the way through to the code that you run.
- Depending on your settings, your source maps might contain an inline copy of your original code. Don't publish them unless you know what you're doing!

CHAPTER 8

Migrating to TypeScript

You've heard that TypeScript is great. You also know from painful experience that maintaining your 15-year-old, 100,000-line JavaScript library isn't. If only it could become a TypeScript library!

This chapter offers some advice about migrating your JavaScript project to TypeScript without losing your sanity and abandoning the effort.

Only the smallest codebases can be migrated in one fell swoop. The key for larger projects is to migrate gradually. Item 60 discusses how to do this. For a long migration, it's essential to track your progress and make sure you don't backslide. This creates a sense of momentum and inevitability to the change. Item 61 discusses ways to do this.

Migrating a large project to TypeScript won't necessarily be easy, but it does offer a huge potential upside. A 2017 study found that 15% of bugs fixed in JavaScript projects on GitHub could have been prevented with TypeScript.[1] Even more impressive, a survey of six months' worth of postmortems at AirBnb found that 38% of them could have been prevented by TypeScript.[2] If you're advocating for TypeScript at your organization, stats like these will help! So will running some experiments and finding early adopters. Item 59 discusses how to experiment with TypeScript before you begin migration.

Since this chapter is largely about JavaScript, many of the code samples are either pure JavaScript (and not expected to pass the type checker) or checked with looser settings (e.g., with `noImplicitAny` off).

1 Z. Gao, C. Bird, and E. T. Barr, "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript," ICSE 2017, *http://earlbarr.com/publications/typestudy.pdf*.

2 Brie Bunge, "Adopting TypeScript at Scale," JSConf Hawaii 2019, *https://youtu.be/P-J9Eg7hJwE*.

Item 58: Write Modern JavaScript

In addition to checking your code for type safety, TypeScript compiles your TypeScript code to any version of JavaScript code, all the way back to 1999 vintage ES3. Since TypeScript is a superset of the *latest* version of JavaScript, this means that you can use `tsc` as a "transpiler": something that takes new JavaScript and converts it to older, more widely supported JavaScript.

Taking a different perspective, this means that when you decide to convert an existing JavaScript codebase to TypeScript, there's no downside to adopting all the latest JavaScript features. In fact, there's quite a bit of upside: because TypeScript is designed to work with modern JavaScript, modernizing your JS is a great first step toward adopting TypeScript.

And because TypeScript is a superset of JavaScript, learning to write more modern and idiomatic JavaScript means you're learning to write better TypeScript, too.

This item gives a quick tour of some of the features in modern JavaScript, which I'm defining here as everything introduced in ES2015 (aka ES6) and after. This material is covered in much greater detail in other books and online. If any of the topics mentioned here are unfamiliar, you owe it to yourself to learn more about them. TypeScript can be tremendously helpful when you're learning a new language feature like `async`/`await`: it almost certainly understands the feature better than you do and can guide you toward correct usage.

These are all worth understanding, but by far the most important for adopting TypeScript are ECMAScript Modules and ES2015 classes.

Use ECMAScript Modules

Before the 2015 version of ECMAScript there was no standard way to break your code into separate modules. There were many solutions, from multiple `<script>` tags, manual concatenation, and Makefiles to node.js-style `require` statements or AMD-style `define` callbacks. TypeScript even had its own module system (Item 53).

Today there is one standard: ECMAScript modules, aka `import` and `export`. If your JavaScript codebase is still a single file, if you use concatenation or one of the other module systems, it's time to switch to ES modules. This may require setting up a tool like webpack or ts-node. TypeScript works best with ES modules, and adopting them will facilitate your transition, not least because it will allow you to migrate modules one at a time (see Item 61).

The details will vary depending on your setup, but if you're using CommonJS like this:

```
// CommonJS
// a.js
const b = require('./b');
console.log(b.name);

// b.js
const name = 'Module B';
module.exports = {name};
```

then the ES module equivalent would look like:

```
// ECMAScript module
// a.ts
import * as b from './b';
console.log(b.name);

// b.ts
export const name = 'Module B';
```

Use Classes Instead of Prototypes

JavaScript has a flexible prototype-based object model. But by and large JS developers have ignored this in favor of a more rigid class-based model. This was officially enshrined into the language with the introduction of the `class` keyword in ES2015.

If your code uses prototypes in a straightforward way, switch to using classes. That is, instead of:

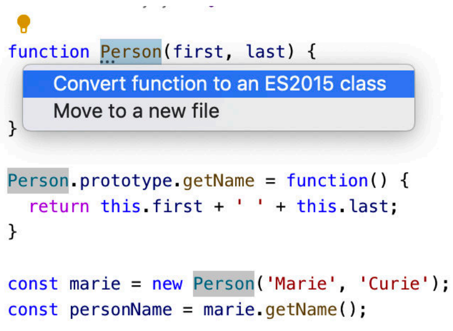
```
function Person(first, last) {
  this.first = first;
  this.last = last;
}

Person.prototype.getName = function() {
  return this.first + ' ' + this.last;
}

const marie = new Person('Marie', 'Curie');
const personName = marie.getName();
```

write:

```
class Person {
  first: string;
  last: string;

  constructor(first: string, last: string) {
    this.first = first;
    this.last = last;
  }

  getName() {
    return this.first + ' ' + this.last;
  }
}

const marie = new Person('Marie', 'Curie');
const personName = marie.getName();
```

TypeScript struggles with the prototype version of `Person` but understands the class-based version with minimal annotations. If you're unfamiliar with the syntax, TypeScript will help you get it right.

For code that uses older-style classes, the TypeScript language service offers a "Convert function to an ES2015 class" quick fix that can speed this up (Figure 8-1).



```
function Person(first, last) {

}

Person.prototype.getName = function() {
  return this.first + ' ' + this.last;
}

const marie = new Person('Marie', 'Curie');
const personName = marie.getName();
```

Figure 8-1. The TypeScript language service offers a quick fix to convert older-style classes to ES2015 classes.

Use let/const Instead of var

JavaScript's `var` has some famously quirky scoping rules. If you're curious to learn more about them, read *Effective JavaScript*. But better to avoid `var` and not worry! Instead, use `let` and `const`. They're truly block-scoped and work in much more intuitive ways than `var`.

Again, TypeScript will help you here. If changing `var` to `let` results in an error, then you're almost certainly doing something you shouldn't be.

Nested function statements also have `var`-like scoping rules:

```
  function foo() {
    bar();
    function bar() {
      console.log('hello');
    }
  }
```

When you call `foo()`, it logs `hello` because the definition of `bar` is hoisted to the top of `foo`. This is surprising! Prefer function expressions (`const bar = () => { ... }`) instead.

Use for-of or Array Methods Instead of for(;;)

In classic JavaScript you used a C-style for loop to iterate over an array:

```
for (var i = 0; i < array.length; i++) {
  const el = array[i];
  // ...
}
```

In modern JavaScript you can use a for-of loop instead:

```
for (const el of array) {
  // ...
}
```

This is less prone to typos and doesn't introduce an index variable. If you want the index variable, you can use `forEach`:

```
array.forEach((el, i) => {
  // ...
});
```

Avoid using the for-in construct to iterate over arrays as it has many surprises (see Item 16).

Prefer Arrow Functions Over Function Expressions

The `this` keyword is one of the most famously confusing aspects of JavaScript because it has different scoping rules than other variables:

```
class Foo {
  method() {
    console.log(this);
    [1, 2].forEach(function(i) {
      console.log(this);
    });
  }
}
const f = new Foo();
f.method();
// Prints Foo, undefined, undefined in strict mode
// Prints Foo, window, window (!) in non-strict mode
```

Generally you want `this` to refer to the relevant instance of whichever class you're in. Arrow functions help you do that by keeping the `this` value from their enclosing scope:

```
class Foo {
  method() {
    console.log(this);
    [1, 2].forEach(i => {
      console.log(this);
    });
  }
}
const f = new Foo();
f.method();
// Always prints Foo, Foo, Foo
```

In addition to having simpler semantics, arrow functions are more concise. You should use them whenever possible. For more on `this` binding, see Item 49. With the `noImplicitThis` (or `strict`) compiler option, TypeScript will help you get your `this`-binding right.

Use Compact Object Literals and Destructuring Assignment

Instead of writing:

```
const x = 1, y = 2, z = 3;
const pt = {
  x: x,
  y: y,
  z: z
};
```

you can simply write:

```
const x = 1, y = 2, z = 3;
const pt = { x, y, z };
```

In addition to being more concise, this encourages consistent naming of variables and properties, something your human readers will appreciate as well (Item 36).

To return an object literal from an arrow function, wrap it in parentheses:

```
['A', 'B', 'C'].map((char, idx) => ({char, idx}));
// [ { char: 'A', idx: 0 },  { char: 'B', idx: 1 }, { char: 'C', idx: 2 } ]
```

There is also shorthand for properties whose values are functions:

```
const obj = {
  onClickLong: function(e) {
    // ...
  },
  onClickCompact(e) {
    // ...
  }
};
```

The inverse of compact object literals is object destructuring. Instead of writing:

```
const props = obj.props;
const a = props.a;
const b = props.b;
```

you can write:

```
const {props} = obj;
const {a, b} = props;
```

or even:

```
const {props: {a, b}} = obj;
```

In this last example only `a` and `b` become variables, not `props`.

You may specify default values when destructuring. Instead of writing:

```
let {a} = obj.props;
if (a === undefined) a = 'default';
```

write this:

```
const {a = 'default'} = obj.props;
```

You can also destructure arrays. This is particularly useful with tuple types:

```
const point = [1, 2, 3];
const [x, y, z] = point;
const [, a, b] = point;  // Ignore the first one
```

Destructuring can also be used in function parameters:

```
const points = [
  [1, 2, 3],
  [4, 5, 6],
];
points.forEach(([x, y, z]) => console.log(x + y + z));
// Logs 6, 15
```

As with compact object literal syntax, destructuring is concise and encourages consistent variable naming. Use it!

Use Default Function Parameters

In JavaScript, all function parameters are optional:

```
function log2(a, b) {
  console.log(a, b);
}
log2();
```

This outputs:

```
undefined undefined
```

This is often used to implement default values for parameters:

```
function parseNum(str, base) {
  base = base || 10;
  return parseInt(str, base);
}
```

In modern JavaScript, you can specify the default value directly in the parameter list:

```
function parseNum(str, base=10) {
  return parseInt(str, base);
}
```

In addition to being more concise, this makes it clear that `base` is an optional parameter. Default parameters have another benefit when you migrate to TypeScript: they help the type checker infer the type of the parameter, removing the need for a type annotation. See Item 19.

Use async/await Instead of Raw Promises or Callbacks

Item 25 explains why `async` and `await` are preferable, but the gist is that they'll simplify your code, prevent bugs, and help types flow through your asynchronous code.

Instead of either of these:

```
function getJSON(url: string) {
  return fetch(url).then(response => response.json());
}
function getJSONCallback(url: string, cb: (result: unknown) => void) {
  // ...
}
```

write this:

```
async function getJSON(url: string) {
  const response = await fetch(url);
  return response.json();
}
```

Don't Put use strict in TypeScript

ES5 introduced "strict mode" to make some suspect patterns more explicit errors. You enable it by putting `'use strict'` in your code:

```
'use strict';
function foo() {
  x = 10;  // Throws in strict mode, defines a global in non-strict.
}
```

If you've never used strict mode in your JavaScript codebase, then give it a try. The errors it finds are likely to be ones that the TypeScript compiler will find, too.

But as you transition to TypeScript, there's not much value in keeping `'use strict'` in your source code. By and large, the sanity checks that TypeScript provides are far stricter than those offered by strict mode.

There is some value in having a `'use strict'` in the JavaScript that `tsc` emits. If you set the `alwaysStrict` or `strict` compiler options, TypeScript will parse your code in strict mode and put a `'use strict'` in the JavaScript output for you.

In short, don't write `'use strict'` in your TypeScript. Use `alwaysStrict` instead.

These are just a few of the many new JavaScript features that TypeScript lets you use. TC39, the body that governs JS standards, is very active, and new features are added year to year. The TypeScript team is currently committed to implementing any feature that reaches stage 3 (out of 4) in the standardization process, so you don't even have to wait for the ink to dry. Check out the TC39 GitHub repo[3] for the latest. As of this writing, the pipeline and decorators proposals in particular have great potential to impact TypeScript.

Things to Remember

- TypeScript lets you write modern JavaScript whatever your runtime environment. Take advantage of this by using the language features it enables. In addition to improving your codebase, this will help TypeScript understand your code.
- Use TypeScript to learn language features like classes, destructuring, and `async`/`await`.
- Don't bother with `'use strict'`: TypeScript is stricter.
- Check the TC39 GitHub repo and TypeScript release notes to learn about all the latest language features.

Item 59: Use @ts-check and JSDoc to Experiment with TypeScript

Before you begin the process of converting your source files from JavaScript to TypeScript (Item 60), you may want to experiment with type checking to get an initial read on the sorts of issues that will come up. TypeScript's `@ts-check` directive lets you do exactly this. It directs the type checker to analyze a single file and report whatever issues it finds. You can think of it as an extremely loose version of type checking: looser even than TypeScript with `noImplicitAny` off (Item 2).

3 *https://github.com/tc39/proposals*

Here's how it works:

```
// @ts-check
const person = {first: 'Grace', last: 'Hopper'};
2 * person.first
 // ~~~~~~~~~~~~ The right-hand side of an arithmetic operation must be of type
 //              'any', 'number', 'bigint', or an enum type
```

TypeScript infers the type of `person.first` as `string`, so `2 * person.first` is a type error, no type annotations required.

While it may surface this sort of blatant type error, or functions called with too many arguments, in practice, `// @ts-check` tends to turn up a few specific types of errors:

Undeclared Globals

If these are symbols that you're defining, then declare them with `let` or `const`. If they are "ambient" symbols that are defined elsewhere (in a `<script>` tag in an HTML file, for instance), then you can create a type declarations file to describe them.

For example, if you have JavaScript like this:

```
// @ts-check
console.log(user.firstName);
         // ~~~~ Cannot find name 'user'
```

then you could create a file called *types.d.ts*:

```
interface UserData {
  firstName: string;
  lastName: string;
}
declare let user: UserData;
```

Creating this file on its own may fix the issue. If it does not, you may need to explicitly import it with a "triple-slash" reference:

```
// @ts-check
/// <reference path="./types.d.ts" />
console.log(user.firstName);  // OK
```

This *types.d.ts* file is a valuable artifact that will become the basis for your project's type declarations.

Unknown Libraries

If you're using a third-party library, TypeScript needs to know about it. For example, you might use jQuery to set the size of an HTML element. With `@ts-check`, TypeScript will flag an error:

```
// @ts-check
   $('#graph').style({'width': '100px', 'height': '100px'});
// ~ Cannot find name '$'
```

The solution is to install the type declarations for jQuery:

```
$ npm install --save-dev @types/jquery
```

Now the error is specific to jQuery:

```
// @ts-check
$('#graph').style({'width': '100px', 'height': '100px'});
         // ~~~~~ Property 'style' does not exist on type 'JQuery<HTMLElement>'
```

In fact, it should be `.css`, not `.style`.

`@ts-check` lets you take advantage of the TypeScript declarations for popular JavaScript libraries without migrating to TypeScript yourself. This is one of the best reasons to use it.

DOM Issues

Assuming you're writing code that runs in a web browser, TypeScript is likely to flag issues around your handling of DOM elements. For example:

```
// @ts-check
const ageEl = document.getElementById('age');
ageEl.value = '12';
   // ~~~~~ Property 'value' does not exist on type 'HTMLElement'
```

The issue is that only `HTMLInputElements` have a `value` property, but `document.getElementById` returns the more generic `HTMLElement` (see Item 55). If you know that the `#age` element really is an `input` element, then this is an appropriate time to use a type assertion (Item 9). But this is still a JS file, so you can't write `as HTMLInputElement`. Instead, you can assert a type using JSDoc:

```
// @ts-check
const ageEl = /** @type {HTMLInputElement} */(document.getElementById('age'));
ageEl.value = '12';  // OK
```

If you mouse over `ageEl` in your editor, you'll see that TypeScript now considers it an `HTMLInputElement`. Take care as you type the JSDoc `@type` annotation: the parentheses after the comment are required.

This leads to another type of error that comes up with `@ts-check`, inaccurate JSDoc, as explained next.

Inaccurate JSDoc

If your project already has JSDoc-style comments, TypeScript will begin checking them when you flip on `@ts-check`. If you previously used a system like the Closure

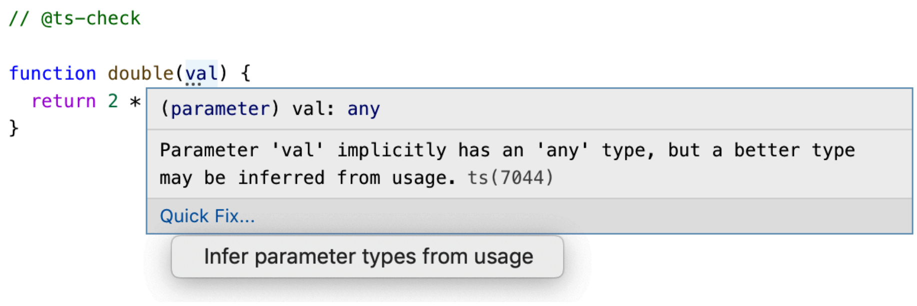
Compiler that used these comments to enforce type safety, then this shouldn't cause major headaches. But you may be in for some surprises if your comments were more like "aspirational JSDoc":

```
// @ts-check
/**
 * Gets the size (in pixels) of an element.
 * @param {Node} el The element
 * @return {{w: number, h: number}} The size
 */
function getSize(el) {
  const bounds = el.getBoundingClientRect();
                 // ~~~~~~~~~~~~~~~~~~~~~ Property 'getBoundingClientRect'
                 //                       does not exist on type 'Node'
  return {width: bounds.width, height: bounds.height};
       // ~~~~~~~~~~~~~~~~~~~ Type '{ width: any; height: any; }' is not
       //                     assignable to type '{ w: number; h: number; }'
}
```

The first issue is a misunderstanding of the DOM: `getBoundingClientRect()` is defined on `Element`, not `Node`. So the `@param` tag should be updated. The second is a mismatch between proprties specified in the `@return` tag and the implementation. Presumably the rest of the project uses the `width` and `height` properties, so the `@return` tag should be updated.

You can use JSDoc to gradually add type annotations to your project. The TypeScript language service will offer to infer type annotations as a quick fix for code where it's clear from usage, as shown here and in Figure 8-2:

```
function double(val) {
  return 2 * val;
}
```



```
// @ts-check

function double(val) {
  return 2 *
}
```

```
(parameter) val: any

Parameter 'val' implicitly has an 'any' type, but a better type
may be inferred from usage. ts(7044)
```

Figure 8-2. The TypeScript Language Services offer a quick fix to infer paramter types from usage.

This results in a correct JSDoc annotation:

```
// @ts-check
/**
 * @param {number} val
 */
function double(val) {
  return 2 * val;
}
```

This can be helpful to encourage types to flow through your code with `@ts-check`. But it doesn't always work so well. For instance:

```
function loadData(data) {
  data.files.forEach(async file => {
    // ...
  });
}
```

If you use the quick fix to annotate `data`, you'll wind up with:

```
/**
 * @param {{
 *  files: { forEach: (arg0: (file: any) => Promise<void>) => void; };
 * }} data
 */
function loadData(data) {
  // ...
}
```

This is structural typing gone awry (Item 4). While the function would technically work on any sort of object with a `forEach` method with that signature, the intent was most likely for the parameter to be `{files: string[]}`.

You can get much of the TypeScript experience in a JavaScript project using JSDoc annotations and `@ts-check`. This is appealing because it requires no changes in your tooling. But it's best not to go too far in this direction. Comment boilerplate has real costs: it's easy for your logic to get lost in a sea of JSDoc. TypeScript works best with *.ts* files, not *.js* files. The goal is ultimately to convert your project to TypeScript, not to JavaScript with JSDoc annotations. But `@ts-check` can be a useful way to experiment with types and find some initial errors, especially for projects that already have extensive JSDoc annotations.

Things to Remember

- Add "`// @ts-check`" to the top of a JavaScript file to enable type checking.
- Recognize common errors. Know how to declare globals and add type declarations for third-party libraries.
- Use JSDoc annotations for type assertions and better type inference.

- Don't spend too much time getting your code perfectly typed with JSDoc. Remember that the goal is to convert to *.ts*!

Item 60: Use allowJs to Mix TypeScript and JavaScript

For a small project, you may be able to convert from JavaScript to TypeScript in one fell swoop. But for a larger project this "stop the world" approach won't work. You need to be able to transition gradually. That means you need a way for TypeScript and JavaScript to coexist.

The key to this is the `allowJs` compiler option. With `allowJs`, TypeScript files and JavaScript files may import one another. For JavaScript files this mode is extremely permissive. Unless you use `@ts-check` (Item 59), the only errors you'll see are syntax errors. This is "TypeScript is a superset of JavaScript" in the most trivial sense.

While it's unlikely to catch errors, `allowJs` does give you an opportunity to introduce TypeScript into your build chain before you start making code changes. This is a good idea because you'll want to be able to run your tests as you convert modules to TypeScript (Item 61).

If your bundler includes TypeScript integration or has a plug-in available, that's usually the easiest path forward. With `browserify`, for instance, you run `npm install --sav-dev tsify` and add it as a plug-in:

```
$ browserify index.ts -p [ tsify --noImplicitAny ] > bundle.js
```

Most unit testing tools have an option like this as well. With the `jest` tool, for instance, you install `ts-jest` and pass TypeScript sources through it by specifying a `jest.config.js` like:

```
module.exports = {
  transform: {
    '^.+\\.tsx?$': 'ts-jest',
  },
};
```

If your build chain is custom, your task will be more involved. But there's always a good fallback option: when you specify the `outDir` option, TypeScript will generate pure JavaScript sources in a directory that parallels your source tree. Usually your existing build chain can be run over that. You may need to tweak TypeScript's JavaScript output so that it closely matches your original JavaScript source, (e.g., by specifying the `target` and `module` options).

Adding TypeScript into your build and test process may not be the most enjoyable task, but it is an essential one that will let you begin to migrate your code with confidence.

Things to Remember

- Use the `allowJs` compiler option to support mixed JavaScript and TypeScript as you transition your project.
- Get your tests and build chain working with TypeScript before beginning large-scale migration.

Item 61: Convert Module by Module Up Your Dependency Graph

You've adopted modern JavaScript, converting your project to use ECMAScript modules and classes (Item 58). You've integrated TypeScript into your build chain and have all your tests passing (Item 60). Now for the fun part: converting your JavaScript to TypeScript. But where to begin?

When you add types to a module, it's likely to surface new type errors in all the modules that depend on it. Ideally you'd like to convert each module once and be done with it. This implies that you should convert modules going *up* the dependency graph: starting with the leaves (modules that depend on no others) and moving up to the root.

The very first modules to migrate are your third-party dependencies since, by definition, you depend on them but they do not depend on you. Usually this means installing `@types` modules. If you use the `lodash` utility library, for example, you'd run `npm install --save-dev @types/lodash`. These typings will help types flow through your code and surface issues in your use of the libraries.

If your code calls external APIs, you may also want to add type declarations for these early on. Although these calls may happen anywhere in your code, this is still in the spirit of moving up the dependency graph since you depend on the APIs but they do not depend on you. Many types flow from API calls, and these are generally difficult to infer from context. If you can find a spec for the API, generate types from that (see Item 35).

As you migrate your own modules, it's helpful to visualize the dependency graph. Figure 8-3 shows an example graph from a medium-sized JavaScript project, made using the excellent `madge` tool.

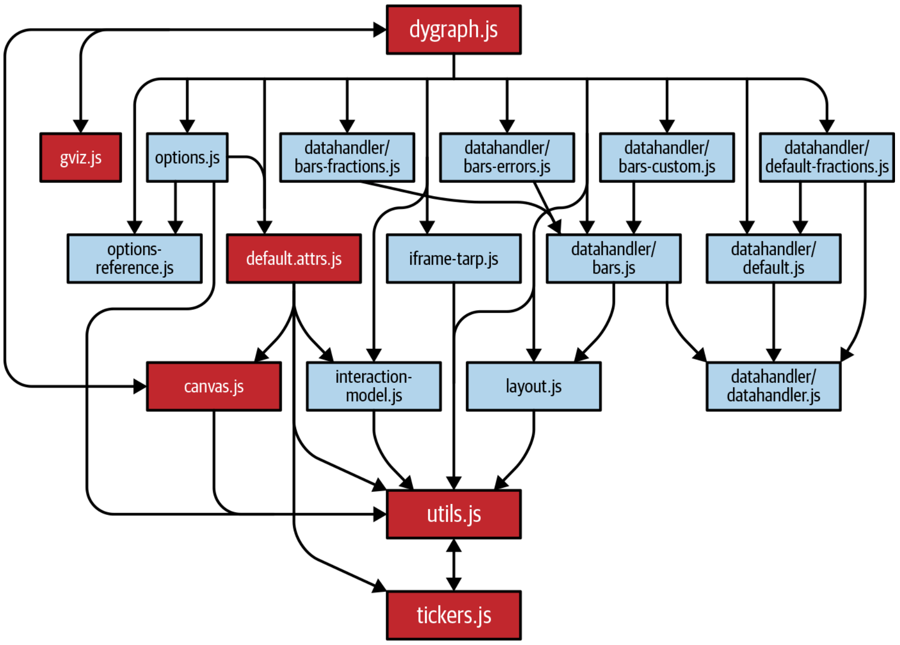


Figure 8-3. The dependency graph for a medium-sized JavaScript project. Arrows indicate dependencies. Darker-shaded boxes indicate that a module is involved in a circular dependency.

The bottom of this dependency graph is the circular dependency between *utils.js* and *tickers.js*. There are many modules that depend on these two, but they only depend on one another. This pattern is quite common: most projects will have some sort of utility module at the bottom of the dependency graph.

As you migrate your code, focus on adding types rather than refactoring. If this is an old project, you're likely to notice some strange things and want to fix them. Resist this urge! The immediate goal is to convert your project to TypeScript, not to improve its design. Instead, write down code smells as you detect them and make a list of future refactors.

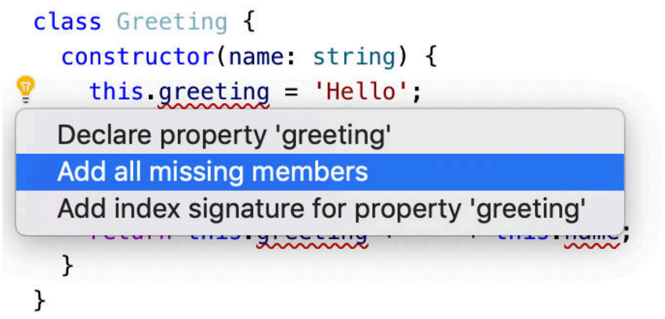
There are a few common errors you'll run into as you convert to TypeScript. Some of these were covered in Item 59, but new ones include:

Undeclared Class Members

Classes in JavaScript do not need to declare their members, but classes in TypeScript do. When you rename a class's *.js* file to *.ts*, it's likely to show errors for every single property you reference:

```
class Greeting {
  constructor(name) {
    this.greeting = 'Hello';
      // ~~~~~~~~ Property 'greeting' does not exist on type 'Greeting'
    this.name = name;
      // ~~~~ Property 'name' does not exist on type 'Greeting'
  }
  greet() {
    return this.greeting + ' ' + this.name;
             // ~~~~~~~~              ~~~~ Property ... does not exist
  }
}
```

There's a helpful quick fix (see Figure 8-4) for this that you should take advantage of.



```
class Greeting {
  constructor(name: string) {
    this.greeting = 'Hello';
```

Declare property 'greeting'
Add all missing members
Add index signature for property 'greeting'

Figure 8-4. The quick fix to add declarations for missing members is particularly helpful in converting a class to TypeScript.

This will add declarations for the missing members based on usage:

```
class Greeting {
  greeting: string;
  name: any;
  constructor(name) {
    this.greeting = 'Hello';
    this.name = name;
  }
  greet() {
    return this.greeting + ' ' + this.name;
  }
}
```

TypeScript was able to get the type for `greeting` correct, but not the type for `name`. After applying this quick fix, you should look through the property list and fix the `any` types.

If this is the first time you've seen the full property list for your class, you may be in for a shock. When I converted the main class in *dygraph.js* (the root module in

Figure 8-3), I discovered that it had no fewer than 45 member variables! Migrating to TypeScript has a way of surfacing bad designs like this that were previously implicit. It's harder to justify a bad design if you have to look at it. But again, resist the urge to refactor now. Note the oddity and think about how you'd fix it some other day.

Values with Changing Types

TypeScript will complain about code like this:

```
const state = {};
state.name = 'New York';
   // ~~~~ Property 'name' does not exist on type '{}'
state.capital = 'Albany';
   // ~~~~~~~ Property 'capital' does not exist on type '{}'
```

This topic is covered in more depth in Item 23, so you may want to brush up on that item if you run into this error. If the fix is trivial, you can build the object all at once:

```
const state = {
  name: 'New York',
  capital: 'Albany',
};  // OK
```

If it is not, then this is an appropriate time to use a type assertion:

```
interface State {
  name: string;
  capital: string;
}
const state = {} as State;
state.name = 'New York';  // OK
state.capital = 'Albany';  // OK
```

You should fix this eventually (see Item 9), but this is expedient and will help you keep the migration going.

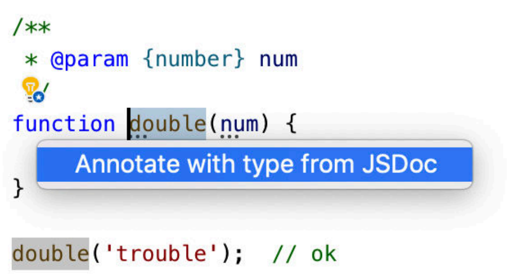
If you've been using JSDoc and `@ts-check` (Item 59), be aware that you can actually *lose* type safety by converting to TypeScript. For instance, TypeScript flags an error in this JavaScript:

```
// @ts-check
/**
 * @param {number} num
 */
function double(num) {
  return 2 * num;
}

double('trouble');
    // ~~~~~~~~~ Argument of type '"trouble"' is not assignable to
    //           parameter of type 'number'
```

When you convert to TypeScript, the `@ts-check` and JSDoc stop being enforced. This means the type of `num` is implicitly `any`, so there's no error:

```
/**
 * @param {number} num
 */
function double(num) {
  return 2 * num;
}

double('trouble');  // OK
```

Fortunately there's a quick fix available to move JSDoc types to TypeScript types. If you have any JSDoc, you should use what's shown in Figure 8-5.



```
/**
 * @param {number} num
 */
function double(num) {

}

double('trouble');  // ok
```

Figure 8-5. Quick fix to copy JSDoc annotations to TypeScript type annotations

Once you've copied type annotations to TypeScript, make sure to remove them from the JSDoc to avoid redundancy (see Item 30):

```
function double(num: number) {
  return 2 * num;
}

double('trouble');
    // ~~~~~~~~~ Argument of type '"trouble"' is not assignable to
    //           parameter of type 'number'
```

This issue will also be caught when you turn on `noImplicitAny`, but you may as well add the types now.

Migrate your tests last. They should be at the top of your dependency graph (since your code doesn't depend on them), and it's extremely helpful to know that your tests continue to pass during the migration despite your not having changed them at all.

Things to Remember

- Start migration by adding `@types` for third-party modules and external API calls.

- Begin migrating your modules from the bottom of the dependency graph upwards. The first module will usually be some sort of utility code. Consider visualizing the dependency graph to help you track progress.
- Resist the urge to refactor your code as you uncover odd designs. Keep a list of ideas for future refactors, but stay focused on TypeScript conversion.
- Be aware of common errors that come up during conversion. Copy JSDoc annotations if necessary to avoid losing type safety as you convert.

Item 62: Don't Consider Migration Complete Until You Enable noImplicitAny

Converting your whole project to *.ts* is a big accomplishment. But your work isn't done quite yet. Your next goal is to turn on the `noImplicitAny` option (Item 2). TypeScript code without `noImplicitAny` is best thought of as transitional because it can mask real errors you've made in your type declarations.

For example, perhaps you've used the "Add all missing members" quick fix to add property declarations to a class (Item 61). You're left with an `any` type and would like to fix it:

```
class Chart {
  indices: any;

  // ...
}
```

`indices` sounds like it should be an array of numbers, so you plug in that type:

```
class Chart {
  indices: number[];

  // ...
}
```

No new errors result, so you then keep moving. Unfortunately, you've made a mistake: `number[]` is the wrong type. Here's some code from elsewhere in the class:

```
getRanges() {
  for (const r of this.indices) {
    const low = r[0];  // Type is any
    const high = r[1];  // Type is any
    // ...
  }
}
```

Clearly `number[][]` or `[number, number][]` would be a more accurate type. Does it surprise you that indexing into a `number` is allowed? Take this as an indication of just how loose TypeScript can be without `noImplicitAny`.

When you turn on `noImplicitAny`, this becomes an error:

```
getRanges() {
  for (const r of this.indices) {
    const low = r[0];
             // ~~~~ Element implicitly has an 'any' type because
             //      type 'Number' has no index signature
    const high = r[1];
              // ~~~~ Element implicitly has an 'any' type because
              //      type 'Number' has no index signature
    // ...
  }
}
```

A good strategy for enabling `noImplicitAny` is to set it in your local client and start fixing errors. The number of errors you get from the type checker gives you a good sense of your progress. You can commit the type corrections without committing the *tsconfig.json* change until you get the number of errors down to zero.

There are many other knobs you can turn to increase the strictness of type checking, culminating with `"strict": true`. But `noImplicitAny` is the most important one and your project will get most of the benefits of TypeScript even if you don't adopt other settings like `strictNullChecks`. Give everyone on your team a chance to get used to TypeScript before you adopt stricter settings.

Things to Remember

- Don't consider your TypeScript migration done until you adopt `noImplicitAny`. Loose type checking can mask real mistakes in type declarations.
- Fix type errors gradually before enforcing `noImplicitAny`. Give your team a chance to get comfortable with TypeScript before adopting stricter checks.

Index

About the Author

Dan Vanderkam is a principal software engineer at Sidewalk Labs. He previously worked on open source genome visualizations at Mount Sinai's Icahn School of Medicine and on Google search features used by billions of people (search for "sunset nyc" or "population of france"). He has a long history of building open source projects and is a cofounder of the NYC TypeScript Meetup.

When he's not programming, Dan enjoys climbing rocks and playing bridge. He writes on Medium and at *danvk.org*. He earned his bachelor's in computer science from Rice University in Houston, Texas, and lives in Brooklyn, New York.

Colophon

The animal on the cover of *Effective TypeScript* is a red-billed oxpecker (*Buphagus erythrorhynchus*). These birds inhabit a fragmented range across eastern Africa, from Ethiopia and Somalia in the northeast to a few pockets in South Africa; however, these birds can be said to inhabit the range of the grazing animals on which they spend almost all their lives.

Red-billed oxpeckers are related to starlings and mynahs, though they are of a distinct and separate family. About eight inches long, and weighing about two ounces, these birds have a bark-brown head, back, and tail, with paler coloring below. Their most striking physical features are their red beaks and red eyes set off by bright yellow eyerings.

Dominating the life of this bird is where and how it finds its food: red-billed oxpeckers feed on ticks and other animal parasites, and they perch on animals as they forage. Their host animals are most often antelope (such as kudu and impala) as well as large animals such as zebra, giraffe, buffalo, and rhinoceros (elephants do not tolerate them). Red-billed oxpeckers have evolved adaptations to assist them in their search for food, such as a flat beak to pierce thick animal hair, and sharp claws and a stiff tail to hang on to their host animals. These birds even conduct courtship while perched on a host animal, and only leave during nesting season. Parent birds raise three chicks in a nest hole (lined with hair pulled from their host) close to the animal herds so that they can feed themselves and their young.

The birds' relationship with their animal hosts was once seen as a clear-cut and classic example of mutualism (a mutually beneficial interaction between species). However, recent studies have shown that oxpeckers' feeding habits don't significantly affect hosts' parasite loads, and additionally showed that oxpeckers actually work to keep animals' wounds open, so that they can feed on their blood.

Red-billed oxpeckers remain common across their range; though pesticide use is a threat, their adoption of domestic cattle herds as a food source helps their population remain stable. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Jose Marzan, based on a black-and-white engraving from *Elements of Ornithology*. The cover fonts are Gilroy and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

Effective TypeScript

TypeScript is a typed superset of JavaScript with the potential to solve many of the headaches for which JavaScript is famous. But TypeScript has a learning curve of its own, and understanding how to use it effectively takes time and practice. Using the format popularized by *Effective C++* and *Effective Java* (both Addison-Wesley), this practical book features 62 items that give specific advice on what to do and what not to do, and how to think about the language.

Author Dan Vanderkam, a principal software engineer at Sidewalk Labs, shows you how to apply each item's advice through concrete examples. This book will help you advance from a beginning or intermediate user familiar with TypeScript basics to an expert who knows how to use the language well.

- **Learn the nuts and bolts of TypeScript's type system**
- **Design types to make your code safer and more understandable**
- **Use type inference to get full safety with a minimum of type annotations**
- **Make tactical use of the *any* type**
- **Understand how dependencies and type declaration files work in TypeScript**
- **Successfully migrate your JavaScript codebase to TypeScript**

Dan Vanderkam is a principal software engineer at Sidewalk Labs, and cofounder of the TypeScript NYC Meetup. A long-time contributor to open source projects, he previously worked at Mount Sinai's Icahn School of Medicine and on search features used by billions of users at Google.

"*Effective TypeScript* explores the most common questions we see when working with TypeScript and provides practical, results-oriented advice. Regardless of your level of TypeScript experience, you can learn something from this book."

—**Ryan Cavanaugh**
Engineering Lead for TypeScript at Microsoft

"This book is packed with practical recipes and must be kept on the desk of every professional TypeScript developer. Even if you think you know TypeScript already, get this book and you won't regret it."

—**Yakov Fain**
Java Champion

PROGRAMMING LANGUAGES / JAVASCRIPT

US $39.99 CAN $52.99
ISBN: 978-1-492-05374-3

5 3 9 9 9
9 781492 053743

Twitter: @oreillymedia
facebook.com/oreilly