Exploring TypeScript

Dr. Axel Rauschmayer

2025-04-21

Copyright © 2025-04-21 by Dr. Axel Rauschmayer

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review or scholarly journal.

exploringjs.com

Table of contents

I Preliminaries 9

1 About this book 11
- 1.1 Where is the homepage of this book? 11
- 1.2 What is in this book? 11
- 1.3 What do I get for my money? 12
- 1.4 How can I preview the content? 12
- 1.5 How do I report errors? 12
- 1.6 What do the notes with icons mean? 12

2 Sales pitch for TypeScript 15
- 2.1 Notation used in this chapter 16
- 2.2 TypeScript benefit: auto-completion and detecting more errors during editing 16
- 2.3 Type annotations for function parameters and results are good documentation 21
- 2.4 TypeScript benefit: better refactoring 22
- 2.5 Using TypeScript has become easier 22
- 2.6 The downsides of using TypeScript 22
- 2.7 TypeScript FAQ 23

3 Free resources on TypeScript 27
- 3.1 Book on JavaScript 27
- 3.2 Books on TypeScript 27
- 3.3 Blogs 27
- 3.4 Coding exercises 28
- 3.5 Other resources 28

II Quick start to TypeScript 29

4 The basics of TypeScript 31
- 4.1 What you'll learn 32
- 4.2 How to play with code while reading this chapter 33
- 4.3 What is a type? 33
- 4.4 TypeScript's two language levels 33

4.5 Primitive literal types . . . 36
4.6 The types any, unknown and never . . . 36
4.7 Type inference . . . 37
4.8 Type aliases . . . 38
4.9 Compound types . . . 38
4.10 Typing Arrays . . . 39
4.11 Function types . . . 40
4.12 Typing objects . . . 42
4.13 Union types . . . 45
4.14 Intersection types . . . 47
4.15 Type guards and narrowing . . . 47
4.16 Type variables and generic types . . . 48
4.17 Conclusion: understanding the initial example . . . 50
4.18 Next steps . . . 51

5 Notation used in this book **53**
5.1 JavaScript level: assert.* . . . 53
5.2 Type level: assertType<T>(v) . . . 54
5.3 Type level: Assert<B> . . . 54
5.4 Type level: @ts-expect-error . . . 55
5.5 Isn't this book's notation kind of ugly? . . . 55

6 How TypeScript is used: workflows, tools, etc. **57**
6.1 TypeScript is JavaScript plus type syntax . . . 58
6.2 Ways of running TypeScript code . . . 58
6.3 Publishing a library package to the npm registry . . . 60
6.4 DefinitelyTyped: a repository with types for type-less npm packages . . . 63
6.5 Compiling TypeScript with tools other than tsc . . . 63
6.6 JSR – the JavaScript registry . . . 65
6.7 Editing TypeScript . . . 65
6.8 Type-checking JavaScript files . . . 66

7 Trying out TypeScript without installing it **67**
7.1 The TypeScript Playground . . . 67
7.2 A simple TypeScript playground via node --watch . . . 68
7.3 Running copied TypeScript code via Node.js . . . 68

III Setting up TypeScript **71**

8 Guide to tsconfig.json **73**
8.1 Features not covered by this chapter . . . 74
8.2 Extending base files via extends . . . 75
8.3 Where are the input files? . . . 75
8.4 What is the output? . . . 75
8.5 Language and platform features . . . 79
8.6 Module system . . . 80
8.7 Type checking . . . 83
8.8 Compiling TypeScript with tools other than tsc . . . 87

8.9 Importing CommonJS from ESM . . . 92
8.10 One more option with a good default . . . 93
8.11 Visual Studio Code . . . 93
8.12 Summary: Assemble your `tsconfig.json` by answering four questions . . 93
8.13 Further reading . . . 96

9 Publishing npm packages with TypeScript **97**
9.1 File system layout . . . 98
9.2 `tsconfig.json` . . . 99
9.3 `package.json` . . . 101
9.4 Linting npm packages . . . 105
9.5 Further reading . . . 106

10 Creating apps with TypeScript **107**
10.1 Writing TypeScript apps for web browsers . . . 107
10.2 Writing TypeScript apps for server-side runtimes . . . 107

11 Documenting TypeScript APIs via doc comments and TypeDoc **109**
11.1 Doc comments . . . 109
11.2 Generating documentation . . . 110
11.3 Referring to parts of files from doc comments . . . 111
11.4 Further reading . . . 112

12 Strategies for migrating to TypeScript **113**
12.1 Strategy: mixed JavaScript/TypeScript code bases . . . 113
12.2 Strategy: adding type information to plain JavaScript files . . . 114
12.3 Strategy: linting before activating a compiler option . . . 114
12.4 Strategy: Too many errors? Use snapshot testing . . . 115
12.5 Tools that help with migrating to TypeScript . . . 115
12.6 Conclusion and further reading . . . 115

IV Basic types **117**

13 What is a type in TypeScript? Two perspectives **119**
13.1 Two questions for each perspective . . . 119
13.2 Dynamic perspective: a type is a set of values . . . 119
13.3 Static perspective: relationships between types . . . 120
13.4 Nominal type systems vs. structural type systems . . . 120
13.5 Further reading . . . 121

14 The top types `any` and `unknown` **123**
14.1 TypeScript's two top types . . . 123
14.2 The top type `any` . . . 123
14.3 The top type `unknown` . . . 125

15 The bottom type `never` **127**
15.1 `never` is a bottom type . . . 127
15.2 `never` is the empty set . . . 128
15.3 Use case for `never`: filtering union types . . . 128

15.4 Use case for `never`: exhaustiveness checks at compile time . . . 129
15.5 Use case for `never`: forbidding properties . . . 131
15.6 Functions that return `never` . . . 131
15.7 Sources of this chapter . . . 132

16 Symbols in TypeScript **133**
16.1 Types for symbols . . . 133
16.2 Unions of symbol types . . . 136
16.3 Symbols as special values . . . 137
16.4 Symbols as enum values . . . 137
16.5 Further reading . . . 138

17 Adding special values to types **139**
17.1 Adding special values in band . . . 139
17.2 Adding special values out of band . . . 141

V Typing objects, classes and Arrays **145**

18 Typing objects **147**
18.1 Object types . . . 148
18.2 Members of object literal types . . . 149
18.3 Excess property checks: When are extra properties allowed? . . . 153
18.4 Object types and inherited properties . . . 158
18.5 Interfaces vs. object literal types . . . 158
18.6 Forbidding properties via `never` . . . 163
18.7 Index signatures: objects as dictionaries . . . 164
18.8 `Record<K, V>` for dictionary objects . . . 167
18.9 `object` vs `Object` vs. `{}` . . . 168
18.10 Summary: `object` vs `Object` vs. `{}` vs. `Record` . . . 172
18.11 Sources of this chapter . . . 173

19 Unions of object types **175**
19.1 From unions of object types to discriminated unions . . . 175
19.2 Deriving types from discriminated unions . . . 180
19.3 Class hierarchies vs. discriminated unions . . . 183
19.4 Defining discriminated unions via classes . . . 185

20 Intersections of object types **187**
20.1 Intersections of object types . . . 187

21 Class definitions in TypeScript **191**
21.1 Cheat sheet: classes in plain JavaScript . . . 192
21.2 Non-public data slots in TypeScript . . . 196
21.3 Private constructors . . . 200
21.4 Initializing instance properties . . . 201
21.5 Convenience features we should avoid . . . 202
21.6 Abstract classes . . . 203
21.7 Keyword `override` for methods . . . 205
21.8 Classes vs. object types . . . 205

22 Class-related types **209**
22.1 The two prototype chains of classes 209
22.2 Interfaces for instances of classes 210
22.3 Interfaces for classes 211
22.4 Classes as types 213
22.5 Further reading 216

23 Types for classes as values **217**
23.1 Question: Which type for a class as a value? 217
23.2 Answer: types for classes as values 218
23.3 A generic type for constructors: `Class<T>` 219

24 Where are the remaining chapters? **225**

- Copyright by Dr. Axel Rauschmayer
- Cover image by pickpik.com

Part I

Preliminaries

Chapter 1

About this book

1.1 Where is the homepage of this book? 11
1.2 What is in this book? 11
1.3 What do I get for my money? 12
1.4 How can I preview the content? 12
1.5 How do I report errors? 12
1.6 What do the notes with icons mean? 12

1.1 Where is the homepage of this book?

The homepage of "Exploring TypeScript" is `exploringjs.com/ts/`

1.2 What is in this book?

The chapters of this book are grouped into *parts*:

- Meta-content (information about the book etc.):
  - Part "Preliminaries"
- Using TypeScript almost immediately:
  - Part "Quick start to TypeScript"
- More information on common TypeScript features:
  - Part "Setting up TypeScript"
  - Part "Basic types"
  - Part "Types for objects, classes, Arrays, and functions"
  - Part "Dealing with ambiguous types"
- Advanced usage of types:
  - Part "Computing with types"

Required knowledge: You must know JavaScript. If you want to refresh your knowledge: My book "Exploring JavaScript" is free to read online.

1.3 What do I get for my money?

If you buy the digital package, you get the book in three DRM-free versions:

- PDF file
- ZIP archive with ad-free HTML
- EPUB file

1.4 How can I preview the content?

On the homepage of this book, there are extensive previews for all versions of this book.

1.5 How do I report errors?

- The HTML version of this book has a link to comments at the end of each chapter.
- They jump to GitHub issues, which you can also access directly.

1.6 What do the notes with icons mean?

Reading instructions

Explains how to best read the content.

External content

Points to additional, external, content.

Tip

Gives a tip related to the current content.

Question

Asks and answers a question pertinent to the current content (think FAQ).

Warning

Warns about pitfalls, etc.

Details

Provides additional details, complementing the current content. It is similar to a footnote.

GitHub repository

Mentions a relevant GitHub repository.

Chapter 2

Sales pitch for TypeScript

2.1 Notation used in this chapter . . . 16
2.2 TypeScript benefit: auto-completion and detecting more errors during editing 16
  2.2.1 Example: typos, incorrect types, missing arguments . . . 16
  2.2.2 Example: getting function results wrong . . . 17
  2.2.3 Example: working with optional properties . . . 18
  2.2.4 Example: forgetting switch cases . . . 18
  2.2.5 Example: code handles some cases incorrectly . . . 20
2.3 Type annotations for function parameters and results are good documentation 21
2.4 TypeScript benefit: better refactoring . . . 22
2.5 Using TypeScript has become easier . . . 22
2.6 The downsides of using TypeScript . . . 22
2.7 TypeScript FAQ . . . 23
  2.7.1 Is TypeScript code heavyweight? . . . 23
  2.7.2 Is TypeScript trying to turn JavaScript into C# or Java? . . . 23
  2.7.3 Advanced usage of types seems very complicated. Do I really have to learn that? . . . 25
  2.7.4 How long does it take to learn TypeScript? . . . 25

Roughly, TypeScript is JavaScript plus type information. The latter is removed before TypeScript code is executed by JavaScript engines. Therefore, writing and deploying TypeScript is more work. Is that added work worth it? In this chapter, I'm going to argue that yes, it is. Read it if you are skeptical about TypeScript but interested in giving it a chance.

You can skip this chapter if you're already sure you want to learn and use TypeScript

2.1 Notation used in this chapter

In TypeScript code, I'll show the errors reported by TypeScript via comments that start with `@ts-expect-error` – e.g.:

```
// @ts-expect-error: The right-hand side of an arithmetic operation
// must be of type 'any', 'number', 'bigint' or an enum type.
const value = 5 * '8';
```

That makes it easier to automatically test all the source code in this chapter. It's also a built-in TypeScript feature that can be useful (albeit rarely).

2.2 TypeScript benefit: auto-completion and detecting more errors during editing

Let's look at examples of code where TypeScript helps us – by auto-completing and by detecting errors. The first example is simple; later ones are more sophisticated.

2.2.1 Example: typos, incorrect types, missing arguments

```
class Point {
  x: number;
  y: number;
  constructor(x: number, y = x) {
    this.x = x;
    this.y = y;
  }
}
const point1 = new Point(3, 8);

// @ts-expect-error: Property 'z' does not exist on type 'Point'.
console.log(point1.z); // (A)

// @ts-expect-error: Property 'toUpperCase' does not exist on
// type 'number'.
point1.x.toUpperCase(); // (B)

const point2 = new Point(3); // (C)

// @ts-expect-error: Expected 1-2 arguments, but got 0.
const point3 = new Point(); // (D)

// @ts-expect-error: Argument of type 'string' is not assignable to
// parameter of type 'number'.
const point4 = new Point(3, '8'); // (E)
```

What is happening here?

- Line A: TypeScript knows the type of `point1` and it doesn't have a property `.z`.

- Line B: `point1.x` is a number and therefore doesn't have the string method `.toUpperCase()`.
- Line C: This invocation works because the second argument of `new Point()` is optional.
- Line D: At least one argument must be provided.
- Line E: The second argument of `new Point()` must be a number.

In line A, we get auto-completion after `point1.` (the properties `x` and `y` of that object):

2.2.2 Example: getting function results wrong

How many issues can you see in the following JavaScript code?

```
function reverseString(str) {
  if (str.length === 0) {
    return str;
  }
  Array.from(str).reverse();
}
```

Let's see what TypeScript tells us if we add type annotations (line A):

```
// @ts-expect-error: Function lacks ending return statement and
// return type does not include 'undefined'.
function reverseString(str: string): string { // (A)
  if (str.length === 0) {
    return str;
  }
  Array.from(str).reverse(); // (B)
}
```

TypeScript tells us:

- At the end, there is no `return` statement – which is true: We forgot to start line B with `return` and therefore implicitly return `undefined` after line B.
- The implicitly returned `undefined`is incompatible with the return type `string` (line A).

If we fix this issue, TypeScript points out another error:

```
function reverseString(str: string): string { // (A)
  if (str.length === 0) {
    return str;
  }
```

```
  // @ts-expect-error: Type 'string[]' is not assignable to
  // type 'string'.
  return Array.from(str).reverse(); // (B)
}
```

In line B, we are returning an Array while the return type in line A says that we want to return a string. If we fix that issue too, TypeScript is finally happy with our code:

```
function reverseString(str: string): string {
  if (str.length === 0) {
    return str;
  }
  return Array.from(str).reverse().join('');
}
```

2.2.3 Example: working with optional properties

In our next example, we work with names that are defined via objects. We define the structure of those objects via the following TypeScrip type:

```
type NameDef = {
  name?: string, // (A)
  nick?: string, // (B)
};
```

In other words: NameDef objects have two properties whose values are strings. Both properties are optional – which is indicated via the question marks in line A and line B.

The following code contains an error and TypeScript warns us about it:

```
function getName(nameDef: NameDef): string {
  // @ts-expect-error: Type 'string | undefined' is not assignable
  // to type 'string'.
  return nameDef.nick ?? nameDef.name;
}
```

`??` is the nullish coalescing operator that returns its left-hand side – unless it is `undefined` or `null`. In that case, it returns its right-hand side. For more information, see "Exploring JavaScript".

`nameDef.name` may be missing. In that case, the result is `undefined` and not a string. If we fix that, TypeScript does not report any more errors:

```
function getName(nameDef: NameDef): string {
  return nameDef.nick ?? nameDef.name ?? '(Anonymous)';
}
```

2.2.4 Example: forgetting switch cases

Consider the following type for colors:

```
type Color = 'red' | 'green' | 'blue';
```

In other words: a color is either the string `'red'` or the string `'green'` or the string `'blue'`. The following function translates such colors to CSS hexadecimal color values:

```
function getCssColor(color: Color): `#${string}` {
  switch (color) {
    case 'red':
      return '#FF0000';
    case 'green':
      // @ts-expect-error: Type '"00FF00"' is not assignable to
      // type '`#${string}`'.
      return '00FF00'; // (A)
    default:
      // (B)
      // @ts-expect-error: Argument of type '"blue"' is not
      // assignable to parameter of type 'never'.
      throw new UnexpectedValueError(color); // (C)
  }
}
```

In line A, we get an error because we return a string that is incompatible with the return type `` `#${string}` ``: It does not start with a hash symbol.

The error in line C means that we forgot a case (the value `'blue'`). To understand the error message, we must know that TypeScript continually adapts the type of `color`:

- Before the `switch` statement, its type is `'red' | 'green' | 'blue'`.
- After we crossed off the cases `'red'` and `'green'`, its type is `'blue'` in line B.

And that type is incompatible with the special type `never` that the parameter of `new UnexpectedValueError()` has. That type is used for variables at locations that we never reach. For more information see "The bottom type `never`" (§15).

After we fix both errors, our code looks like this:

```
function getCssColor(color: Color): `#${string}` {
  switch (color) {
    case 'red':
      return '#FF0000';
    case 'green':
      return '#00FF00';
    case 'blue':
      return '#0000FF';
    default:
      throw new UnexpectedValueError(color);
  }
}
```

This is what the error class `UnexpectedValueError` looks like:

```
class UnexpectedValueError extends Error {
  constructor(
    // Type enables type checking
```

```
    value: never,
    // Avoid exception if `value` is:
    // - object without prototype
    // - symbol
    message = `Unexpected value: ${{}.toString.call(value)}`
  ) {
    super(message)
  }
}
```

Lastly, TypeScript gives us auto-completion for the argument of `getCssColor()` (the values `'blue'`, `'green'` and `'red'` that we can use for it):

2.2.5 Example: code handles some cases incorrectly

The following type describes content via objects. Content can be text, an image or a video:

```
type Content =
  | {
    kind: 'text',
    charCount: number,
  }
  | {
    kind: 'image',
    width: number,
    height: number,
  }
  | {
    kind: 'video',
    width: number,
    height: number,
    runningTimeInSeconds: number,
  }
;
```

In the following code, we use content incorrectly:

```
function getWidth(content: Content): number {
  // @ts-expect-error: Property 'width' does not exist on
  // type 'Content'.
  return content.width;
}
```

TypeScript warns us because not all kinds of content have the property `.content`. However, they all do have the property `.kind` – which we can use to fix the error:

```
function getWidth(content: Content): number {
  if (content.kind === 'text') {
    return NaN;
  }
  return content.width; // (A)
}
```

Note that TypeScript does not complain in line A, because we have excluded text content, which is the only content that does not have the property `.width`.

2.3 Type annotations for function parameters and results are good documentation

Consider the following JavaScript code:

```
function filter(items, callback) {
  // ···
}
```

That does not tell us much about the arguments expected by `filter()`. We also don't know what it returns. In contrast, this is what the corresponding TypeScript code looks like:

```
function filter(
  items: Iterable<string>,
  callback: (item: string, index: number) => boolean
): Iterable<string> {
  // ···
}
```

This information tells us:

- Argument `items` is an iterable over strings.
- The `callback` receives a string and an index and returns a boolean.
- The result of `filter()` is another iterable over strings.

Yes, the type notation takes getting used to. But, once we understand it, we can quickly get a rough understand of what `filter()` does. More quickly than by reading prose in English (which, admittedly, is still needed to fill in the gaps left by the type notation and the name of the function).

I find it easier to understand TypeScript code bases than JavaScript code bases because, to me, TypeScript provides an additional layer of documentation.

This additional documentation also helps when working in teams because it is clearer how code is to be used and TypeScript often warns us if we are doing something wrong.

Whenever I migrate JavaScript code to TypeScript, I'm noticing an interesting phenomenon: In order to find the appropriate types for the parameters of a function or method, I have

to check where it is invoked. That means that static types give me information locally that I otherwise have to look up elsewhere.

2.4 TypeScript benefit: better refactoring

Refactorings are automated code transformations that many integrated development environments offer.

Renaming methods is an example of a refactoring. Doing so in plain JavaScript can be tricky because the same name might refer to different methods. TypeScript has more information on how methods and types are connected, which makes renaming methods safer there.

2.5 Using TypeScript has become easier

We now often don't need an extra build step compared to JavaScript:

- On server side JavaScript platforms such as Node.js, Deno and Bun, we can run TypeScript directly – without compiling it.
- Most bundlers such as Vite have built-in support for TypeScript.

More good news:

- Compiling TypeScript to JavaScript has become more efficient – thanks to a technique called *type stripping* which simply removes the type part of TypeScript syntax and makes no other transformations (more information).

Creating packages has also improved:

- npm: Non-library packages can be published in TypeScript. Library packages must contain JavaScript plus *declaration files* (with type information). Generating the latter has also improved – thanks to a technique called *isolated declarations*.
- JSR (JavaScript Registry) is an alternative to npm where packages can be uploaded as TypeScript. It supports a variety of platforms. For Node.js, it automatically generates JavaScript files and declaration files.

Alas, type checking is still relatively slow and must be performed via the TypeScript compiler `tsc`.

2.6 The downsides of using TypeScript

- It is an added layer on top of JavaScript: more complexity, more things to learn, etc.
- npm packages can only be used if they have static type definitions. These days, most packages either come with type definitions or there are type definitions available for them on DefinitelyTyped. However, especially the latter can occasionally be slightly wrong, which leads to issues that you don't have without static typing.
- Configuring TypeScript via `tsconfig.json` also adds a bit of complexity and means that there is a lot of variation w.r.t. how TypeScript code bases are type-checked. There are two mitigating factors:

- For my own projects, I'm now using a maximally strict `tsconfig.json` – which eliminated my doubts about what my `tsconfig.json` should look like.
- Type stripping (see previous section) has clarified the role of `tsconfig.json` for me: With them, it only configures how type checking works. Generating JavaScript can be done without `tsconfig.json`.

2.7 TypeScript FAQ

2.7.1 Is TypeScript code heavyweight?

TypeScript code *can* be heavyweight. But it doesn't have to be. For example, due to type inference, we can often get away with relatively few type annotations:

```
function setDifference<T>(set1: Set<T>, set2: Set<T>): Set<T> {
  const result = new Set<T>();
  for (const elem of set1) {
    if (!set2.has(elem)) {
      result.add(elem);
    }
  }
  return result;
}
```

The only non-JavaScript syntax in this code is `<T>`: Its first occurrence `setDifference<T>` means that the function `setDifference()` has a *type parameter* – a parameter at the type level. All later occurrences of `<T>` refer to that parameter. They mean:

- The parameters `set1` and `set2` are Sets whose elements have the same type `T`.
- The result is also a Set. Its elements have the same type as those of `set1` and `set2`.

Note that we normally don't have to provide the type parameter `<T>` – TypeScript can extract it automatically from the types of the parameters:

```
assert.deepEqual(
  setDifference(new Set(['a', 'b']), new Set(['b'])),
  new Set(['a']),
);
assert.deepEqual(
  setDifference(new Set(['a', 'b']), new Set(['a', 'b'])),
  new Set(),
);
```

When it comes to *using* `setDifference()`, the TypeScript code is not different from JavaScript code in this case.

2.7.2 Is TypeScript trying to turn JavaScript into C# or Java?

Over time, the nature of TypeScript has evolved.

TypeScript 0.8 was released in October 2012 when JavaScript had remained stagnant for a long time. Therefore, TypeScript added features that its team felt JavaScript was missing -

e.g. classes, modules and enums.

Since then, JavaScript has gained many new features. TypeScript now tracks what JavaScript provides and does not introduce new language-level features anymore – for example:

- In 2012, TypeScript had its own way of doing modules. Now it supports ECMAScript modules and CommonJS.
- In 2012, TypeScript had classes that were transpiled to functions. Since ECMAScript 6 came out in 2015, TypeScript has supported the built-in classes.
- In 2015, TypeScript introduced its own flavor of decorators, in order to support Angular. In 2022, ECMAScript decorators reached stage 3 and TypeScript has supported them since. For more information, see section "The history of decorators" in the 2ality post on ECMAScript decorators.
- If the type checking option `erasableSyntaxOnly` is active, TypeScript only supports JavaScript's language features – e.g. we are not allowed to use enums. This option enables type stripping and is popular among TypeScript programmers. Thus it looks like in the future, most TypeScript will really be pure JavaScript plus type information.
- TypeScript will only get better enums or pattern matching if and when JavaScript gets them.

TypeScript is more than OOP

A common misconception is that TypeScript only supports a class-heavy OOP style; it supports many functional programming patterns just as well – e.g. *discriminated unions* which are a (slightly less elegant) version of algebraic data types:

```
type Content =
  | {
    kind: 'text',
    charCount: number,
  }
  | {
    kind: 'image',
    width: number,
    height: number,
  }
  | {
    kind: 'video',
    width: number,
    height: number,
    runningTimeInSeconds: number,
  }
;
```

In Haskell, this data type would look like this (without labels, for simplicity's sake):

```
data Content =
  Text Int
```

```
  | Image Int Int
  | Video Int Int Int
```

More information: "TypeScript for functional programmers" in the TypeScript Handbook.

2.7.3 Advanced usage of types seems very complicated. Do I really have to learn that?

Normal use of TypeScript almost always involves relatively simple types. For libraries, complicated types can be useful but then they are complicated to write and not complicated to use. My general recommendation is to make types as simple as possible and therefore easier to understand and maintain. If types for code are too complicated then it's often possible to simplify them – e.g. by changing the code and using two functions instead of one or by not capturing every last detail with them.

One key insight for making sense of advanced types, is that they are mostly like a new programming language at the type level and usually describe how input types are transformed into output types. In many ways, they are similar to JavaScript. There are:

- Variables (type variables)
- Functions with parameters (generic types with type parameters)
- Conditional expressions `C ? T : F` (conditional types)
- Loops over objects (mapped types)
- Etc.

For more information on this topic, see "Overview: computing with types".

Are complicated types worth it?

Sometimes they are – for example, as an experiment, I wrote a simple SQL API that gives you a lot of type completions and warnings during editing (if you make typos etc). Note that writing that API involved some work; using it is simple.

2.7.4 How long does it take to learn TypeScript?

I believe that you can learn the basics of TypeScript within a day and be productive the next day. There is still more to learn after that, but you can do so while already using it.

"The basics of TypeScript" (§4) teaches you those basics. If you are new to TypeScript, I'd love to hear from you: Is my assumption correct? Were you able to write (simple) TypeScript after reading it?

Chapter 3

Free resources on TypeScript

3.1 Book on JavaScript 27
3.2 Books on TypeScript 27
3.3 Blogs 27
3.4 Coding exercises 28
3.5 Other resources 28

3.1 Book on JavaScript

- If you see a JavaScript feature in this book that you don't understand, you can look it up in my book "Exploring JavaScript" which is free to read online. Some of the "Further reading" sections at the ends of chapters refer to this book.

3.2 Books on TypeScript

- The official "TypeScript Handbook" is a good reference for the language. It currently has a few holes that are filled by the release notes and GitHub pull requests (which handbook and release notes link to).
- "TypeScript Deep Dive" by Basarat Ali Syed was not updated much after 2020 – e.g., it does not cover template string types. But this book is still a valuable resource.
- "The Concise TypeScript Book" by Simone Poggiali
- "Total TypeScript: Essentials" by Matt Pocock

3.3 Blogs

- My blog "2ality" is about TypeScript and JavaScript.

- Stefan Baumgartner's website "oida.dev" has articles on TypeScript and Rust.
- Josh Goldberg publishes articles about TypeScript.
- Matt Pocock publishes "Articles" and "Tips".

3.4 Coding exercises

- "`Type<Challenge[]>`" by Anthony Fu is a "collection of TypeScript type challenges" – think coding exercises that you solve in TypeScript playgrounds.
- "TypeHero": Coding challenges in playgrounds that come with explanations of the features that are involved.
- "Free TypeScript Tutorials" by Matt Pocock: "A collection of free, exercise-driven, in-depth TypeScript tutorials for you to use on your journey to TypeScript wizardry."
- "Projects" by Josh Goldberg: "Hands on real world projects that will help you exercise your knowledge of TypeScript."

3.5 Other resources

- The TypeScript repository has type definitions for the complete ECMAScript standard library. Reading them is an easy way of practicing TypeScript's type notation.

Part II

Quick start to TypeScript

Chapter 4

The basics of TypeScript

4.1 What you'll learn . . . 32
4.2 How to play with code while reading this chapter . . . 33
4.3 What is a type? . . . 33
4.4 TypeScript's two language levels . . . 33
4.4.1 Dynamic types vs. static types . . . 33
4.4.2 JavaScript's dynamic types . . . 34
4.4.3 TypeScript's static types . . . 34
4.4.4 Revisiting the two language levels . . . 35
4.5 Primitive literal types . . . 36
4.6 The types `any`, `unknown` and `never` . . . 36
4.6.1 The wildcard type `any` . . . 37
4.7 Type inference . . . 37
4.7.1 The rules of type inference . . . 38
4.8 Type aliases . . . 38
4.9 Compound types . . . 38
4.10 Typing Arrays . . . 39
4.10.1 Array types: `T[]` and `Array<T>` . . . 39
4.10.2 Tuple types: `[T0, T1, ···]` . . . 39
4.11 Function types . . . 40
4.11.1 Inferring function types . . . 40
4.11.2 Example: a function whose parameter is a function . . . 40
4.11.3 Inferring the return types of functions . . . 41
4.11.4 The special return type `void` . . . 41
4.11.5 Optional parameters . . . 41
4.11.6 Parameter default values . . . 41
4.11.7 Rest parameters . . . 42
4.12 Typing objects . . . 42
4.12.1 Typing fixed-layout objects via object literal types . . . 42
4.12.2 Interfaces as an alternative to object literal types . . . 43
4.12.3 TypeScript's structural typing vs. nominal typing . . . 43

4.12.4 Optional properties . . . 43
4.12.5 Methods . . . 44
4.13 Union types . . . 45
4.13.1 Adding `undefined` and `null` to types . . . 45
4.13.2 Unions of string literal types . . . 46
4.14 Intersection types . . . 47
4.15 Type guards and narrowing . . . 47
4.16 Type variables and generic types . . . 48
4.16.1 Example: a container for values . . . 48
4.16.2 Example: a generic class . . . 48
4.16.3 Example: Maps . . . 49
4.16.4 Functions and methods with type parameters . . . 49
4.17 Conclusion: understanding the initial example . . . 50
4.18 Next steps . . . 51
4.18.1 Tip: Use `strict` type checking whenever you can . . . 51

This chapter explains the basics of TypeScript. After reading it, you should be able to write your first TypeScript code. My hope is that that shouldn't take you longer than a day. I'd love to hear how long it actually took you – my guess may be off.

Start reading here

You can start reading this book with this chapter: No prior knowledge is required other than JavaScript. Alternatively, if you first want to get a better understanding of how TypeScript fits into development workflows as a tool, you can check out "How TypeScript is used: workflows, tools, etc." (§6).

4.1 What you'll learn

After reading this chapter, you should be able to understand the following TypeScript code (which we'll get back to at the end):

```
interface Array<T> {
  concat(...items: Array<T[] | T>): T[];
  reduce<U>(
    callback: (state: U, element: T, index: number) => U,
    firstState?: U
  ): U;
  // ···
}
```

You may think that this is cryptic. And I agree with you! But (as I hope to prove) this syntax is relatively easy to learn. And once you understand it, it gives you immediate, precise and comprehensive summaries of how code behaves – without having to read long descriptions in English.

4.2 How to play with code while reading this chapter

This chapter is meant to be consumed passively: Everything you need to see is shown here, including explorations of what a piece of code does.

However, you may still want to play with TypeScript code. The following chapter explains how to do that: "Trying out TypeScript without installing it" (§7).

4.3 What is a type?

In this chapter:

- A type is a set of values. For example, the type `boolean` is a set whose elements are `false` and `true`.
- `S` being a subtype of `T` means that `S` is a subset of `T`.

4.4 TypeScript's two language levels

TypeScript is JavaScript plus syntax for adding static type information. Therefore, TypeScript has two *language levels* – two ways of using source code:

- The *program level* (JavaScript): At this level, using TypeScript source code means running it: We have to remove the type information and feed it to a JavaScript engine.
- The *type level* (TypeScript): At this level, using TypeScript source code means type-checking it: We analyze the source code to make sure types are used consistently.

	Program level	**Type level**
Programming language is	JavaScript	TypeScript
Source code is	executed	type-checked
Types are	dynamic	static
Types exist at	runtime	compile time

4.4.1 Dynamic types vs. static types

So far, we have only talked about TypeScript's (static) types. But JavaScript also has types:

```
> typeof true
'boolean'
```

Its types are called *dynamic*. Why is that? We have to run code to see if they are used correctly – e.g.:

```
const value = null;
assert.throws(
  () => value.length,
  /^TypeError: Cannot read properties of null/
);
```

In contrast, TypeScript's types are *static*: We check them by analyzing the syntax – without running the code. That happens during editing (for individual files) or when running the TypeScript compiler `tsc` (for the whole code base). In the following code, TypeScript detects the error via type checking (note that it doesn't even need explicit type information in this case):

```
const value = null;
// @ts-expect-error: 'value' is possibly 'null'.
value.length;
```

`@ts-expect-error` shows type checking errors

In this book, type checking errors are shown via `@ts-expect-error` directives (more information).

4.4.2 JavaScript's dynamic types

The JavaScript language (not TypeScript!) has only eight types. In the ECMAScript specification, they have names that start with capital letters. Here, I'm going with the values returned by `typeof` – e.g.:

```
> typeof undefined
'undefined'
> typeof 123
'number'
> typeof 'abc'
'string'
```

JavaScript's eight types are:

1. `undefined`: the set with the only element `undefined`
2. `null`: the set with the only element `null`. Due to a historical bad decision, `typeof` returns `'object'` for the value `null` and not `'null'`.
3. `boolean`: the set with the two elements `false` and `true`
4. `number`: the set of all numbers
5. `bigint`: the set of all arbitrary-precision integers
6. `string`: the set of all strings
7. `symbol`: the set of all symbols
8. `object`: the set of all objects (which includes functions and Arrays)

`typeof` additionally has a separate "type" for functions but that is not how ECMAScript sees things internally.

All of these types are dynamic. They can also be used at the type level in TypeScript (see next section).

4.4.3 TypeScript's static types

TypeScript brings an additional layer to JavaScript: *static types*. In source code, there are:

- Sources of data – e.g. values created via literals such as `128`, `true` or `['a', 'b']`
- Sinks of data – e.g. storage locations such as variables, properties and parameters.
  - Storage locations can also become data sources when we read from them.

Both have static types in TypeScript:

- The type of a data source describes what dynamic values it can be.
- The type of a data sink describes what dynamic values can be written to it.

One way in which a storage location such as a variable can receive a static type is via a *type annotation* – e.g.:

```
let count: number;
```

The colon (`:`) plus the type `number` is the type annotation. It states that the static type of the variable `count` is `number`. The type annotation helps with type checking:

```
let count: number;
// @ts-expect-error: Type 'string' is not assignable to type 'number'.
count = 'yes';
```

What does the error message mean? The (implicit) static type `string` of the data source `'yes'` is incompatible with the (explicitly specified) static type `number` of the data sink `count`.

A function with type annotations

The next example shows a function with type annotations:

```
function toString(num: number): string {
  return String(num);
}
```

There are two type annotations:

- The parameter `num` has the type `number`.
- The return type of the function is `string`.

4.4.4 Revisiting the two language levels

Let's briefly revisit the two language levels. It's interesting to see how they show up in TypeScript's syntax:

```
const noValue: undefined = undefined;
```

- At the dynamic level, we use JavaScript to declare a variable `noValue` and initialize it with the value `undefined`.

- At the static level, we use TypeScript to specify that variable `noValue` has the static type `undefined`.

The same syntax, `undefined`, is used at the JavaScript level and at the type level and means different things – depending on where it is used.

4.5 Primitive literal types

Several primitive types have so-called *literal types*:

```
let thousand: 1000 = 1000;
```

The `1000` after the colon is a type, a *number literal type*: It is a set whose only element is the value `1000` and it is a subtype of `number`.

On one hand, any value we assign to `thousand` must be `1000`:

```
thousand = 1000; // OK
// @ts-expect-error: Type '999' is not assignable to type '1000'.
thousand = 999;
```

On the other hand, we can assign `thousand` to any variable whose type is `number` because its type is a subtype of `number`:

```
const num: number = thousand;
```

Except for `symbol`, all primitive types have literal types:

```
// boolean literal type
const TRUTHY: true = true;

// bigint literal type
const HUNDRED: 100n = 100n;

// string literal type
const YES: 'yes' = 'yes';

// These could also be considered literal types
const UNDEF: undefined = undefined;
const NULL: null = null;
```

Especially string literal types will become useful later (when we get to union types).

4.6 The types `any`, `unknown` and `never`

TypeScript has several types that are specific to the type level:

- `any` is a wildcard type and accepts any value (see below).
- `unknown` is similar to `any` but less flexible: If a variable or parameter has that type, we can also write any value to it. However, we can't do anything with its content unless we perform further type checks. Being less flexible is a good thing: I recommend avoiding `any` and instead using `unknown` whenever possible. For more information see "The top types `any` and `unknown`" (§14).
- `never` the empty set as a type. Among other things, it is used for locations that are never reached when a program is executed.

4.6.1 The wildcard type `any`

If the type of a storage location is neither explicitly specified nor inferrable, TypeScript uses the type `any` for it. `any` is the type of all values and a wildcard type: If a value has that type, TypeScript does not limit us in any way.

If `strict` type checking is enabled, we can only use `any` explicitly: Every location must have an explicit or inferred static type. That is safer because there are no holes in type checking, no unintended blind spots.

Let's look at examples – the type of parameters can usually not be inferred:

```
// @ts-expect-error: Parameter 'arg' implicitly has an 'any' type.
function func1(arg): void {} // error

function func2(arg: boolean): void {} // OK

function func3(arg = false): void {} // OK
```

For `func3`, TypeScript can infer that `arg` has the type `boolean` because it has the default value `false`.

4.7 Type inference

In many cases, TypeScript can automatically derive the types of data sources or data sinks, without us having to annotate anything. That is called *type inference*.

This is an example of type inference:

```
const count = 14;
assertType<14>(count);
```

`assertType<T>(v)` shows the type `T` of a value `v`

In this book, `assertType<T>(v)` is used to show that a value `v` has the type `T` – which was either inferred or explicitly assigned. For more information see "Type level: `assertType<T>(v)`" (§5.2).

TypeScript infers that the type of `count` is `14`. It can do so because it knows that the value `14` has the type `14`. Interestingly, TypeScript infers a more general type when we use `let`:

```
let count = 14;
assertType<number>(count);
```

Why is that? The assumption is that the value of `count` is preliminary and that we want to assign other (similar!) values later on. If `count` had the type `14` then we wouldn't be able to do that.

Another example of type inference: In this case TypeScript infers that function `toString()` has the return type `string`.

```
function toString(num: number) {
  return String(num);
}
assertType<string>(toString(32));
```

4.7.1 The rules of type inference

Type inference is not guesswork: It follows clear rules (similar to arithmetic) for deriving types where they haven't been specified explicitly. For example:

```
const strValue = String(32);
assertType<string>(strValue);
```

The inferred type of `strValue` is `string`:

Step 1: The inferred type of `32` is `32`.

Step 2: `String` used as a function has the following type (simplified):

```
(value: any) => string
```

This type notation is used for functions and means:

- The function has one parameter, `value`. That parameter has the type `any`. If a parameter has that type, it accepts any kind of value. (More on `any` soon.)
- The function returns values of type `string`.

Step 3: By combining the results of step 1 and step 2, TypeScript can infer that `strValue` has the type `string`.

4.8 Type aliases

With `type` we can create a new name (an alias) for an existing type:

```
type Age = number;
const age: Age = 82;
```

4.9 Compound types

Compound types have other types inside them – which makes them very expressive. These are a few examples:

```
// Array types
type StringArray = Array<string>;

// Function types
type NumToStr = (num: number) => string;

// Object literal types
type BlogPost = {
  title: string,
  tags: Array<string>,
```

```
};

// Union types
type YesOrNo = 'yes' | 'no';
```

Next, we'll explore all of these compound types and more.

4.10 Typing Arrays

TypeScript has two different ways of typing Arrays:

- An *Array type* `T[]` or `Array<T>` is used if an Array is a collection of values that all have the same type `T`.
- A *tuple type* `[T0, T1, ···]` is used if the index of an Array element determines its type.

4.10.1 Array types: `T[]` and `Array<T>`

For historical reasons, there are two equivalent ways of expressing the fact that `arr` is an Array, used to manage a sequence of numbers (think list, stack, queue, etc.):

```
let arr1: number[] = [];
let arr2: Array<number> = [];
```

Normally, TypeScript can infer the type of a variable if there is an assignment. In this case, we have to help it because with an empty Array, it can't determine the type of the elements.

We'll explore the angle brackets notation of `Array<number>` in more detail later (spoiler: `Array` is a *generic type* and `number` is a *type parameter*).

In JavaScript's standard library, `Object.keys()` returns an array:

```
const keys = Object.keys({prop: 123});
assertType<string[]>(keys);
```

4.10.2 Tuple types: `[T0, T1, ···]`

The following variable `entry` has a tuple type:

```
const entry: [string, number] = ['count', 33];
```

We can use it to create an object via `Object.fromEntries()`:

```
assert.deepEqual(
  Object.fromEntries([entry]),
  {
    count: 33,
  }
);
```

What is the nature of `entry`? At the JavaScript level, it's also an Array, but it is used differently:

- The length is fixed: `2`.

- The index of an element determines its meaning and its type:
  - At index 0, there are keys whose type is `string`.
  - At index 1, there are values whose type is `number`.

4.11 Function types

This is an example of a function type:

```
type NumToStr = (num: number) => string;
```

This type comprises every function that accepts a single parameter of type number and returns a string. Let's use this type in a type annotation:

```
const toString: NumToStr = (num) => String(num);
```

Because TypeScript knows that `toString` has the type `NumToStr`, we do not need type annotations inside the arrow function.

4.11.1 Inferring function types

We can also define `toString` like this:

```
const toString = (num: number): string => String(num);
```

Note that we specified both a type for the parameter `num` and a return type. The inferred type of `toString` is:

```
assertType<
  (num: number) => string
>(toString);
```

4.11.2 Example: a function whose parameter is a function

The following function has a parameter `callback` whose type is a function:

```
function stringify123(callback: (num: number) => string): string {
  return callback(123);
}
```

Due to the type of the parameter `callback`, TypeScript rejects the following function call:

```
// @ts-expect-error: Argument of type 'NumberConstructor' is not
// assignable to parameter of type '(num: number) => string'.
stringify123(Number);
```

But it accepts this function call:

```
assert.equal(
  stringify123(String), '123'
);
```

We can also use an arrow function to implement `stringify123()`:

```
const stringify123 =
  (callback: (num: number) => string): string => callback(123);
```

4.11.3 Inferring the return types of functions

TypeScript is good at inferring the return types of functions, but specifying them explicitly is recommended: It makes intentions clearer, enables additional consistency checks and helps external tools with generating declaration files (those tools usually can't infer return types).

4.11.4 The special return type `void`

`void` is a special return type for a function: It tells TypeScript that the function always returns `undefined`.

It may do so explicitly:

```
function f1(): void {
  return undefined;
}
```

Or it may do so implicitly:

```
function f2(): void {}
```

However, such a function cannot explicitly return values other than `undefined`:

```
function f3(): void {
  // @ts-expect-error: Type 'string' is not assignable to type 'void'.
  return 'abc';
}
```

4.11.5 Optional parameters

A question mark after an identifier means that the parameter is optional. For example:

```
function stringify123(callback?: (num: number) => string) {
  if (callback === undefined) {
    callback = String;
  }
  return callback(123); // (A)
}
```

TypeScript only lets us make the function call in line A if we make sure that `callback` isn't `undefined` (which it is if the parameter was omitted).

4.11.6 Parameter default values

TypeScript supports parameter default values:

```
function createPoint(x=0, y=0): [number, number] {
  return [x, y];
}

assert.deepEqual(
  createPoint(),
```

```
  [0, 0]);
assert.deepEqual(
  createPoint(1, 2),
  [1, 2]);
```

Default values make parameters optional. We can usually omit type annotations, because TypeScript can infer the types. For example, it can infer that `x` and `y` both have the type `number`.

If we wanted to add type annotations, that would look as follows.

```
function createPoint(x:number = 0, y:number = 0): [number, number] {
  return [x, y];
}
```

4.11.7 Rest parameters

We can also use rest parameters in TypeScript parameter definitions. Their static types must be Arrays or tuples:

```
function joinNumbers(...nums: number[]): string {
  return nums.join('-');
}
assert.equal(
  joinNumbers(1, 2, 3),
  '1-2-3'
);
```

4.12 Typing objects

Similarly to Arrays, objects can be used in two ways in JavaScript (that are occasionally mixed):

- Fixed-layout object: A fixed number of properties that are known at development time. Each property can have a different type.
- Dictionary object: An arbitrary number of properties whose names are not known at development time. All properties have the same type.

We are ignoring dictionary objects in this chapter – they are covered in "Index signatures: objects as dictionaries" (§18.7). As an aside, Maps are usually a better choice for dictionaries, anyway.

4.12.1 Typing fixed-layout objects via object literal types

Object literal types describe fixed-layout objects – e.g.:

```
type Point = {
  x: number,
  y: number,
};
```

We can also use semicolons instead of commas to separate members, but the latter are more common.

The members can also be separated by semicolons instead of commas but since the syntax of object literals types is related to the syntax of object literals (where members must be separated by commas), commas are used more often.

4.12.2 Interfaces as an alternative to object literal types

Interfaces are mostly equivalent to object literal types but have become less popular over time. This is what an interface looks like:

```
interface Point {
  x: number;
  y: number;
} // no semicolon!
```

The members can also be separated by commas instead of semicolons but since the syntax of interfaces is related to the syntax of classes (where members must be separated by semicolons), semicolons are used more often.

4.12.3 TypeScript's structural typing vs. nominal typing

One big advantage of TypeScript's type system is that it works *structurally*, not *nominally*. That is, the type `Point` matches all objects that have the appropriate structure:

```
type Point = {
  x: number,
  y: number,
};
function pointToString(pt: Point) {
  return `(${pt.x}, ${pt.y})`;
}

assert.equal(
  pointToString({x: 5, y: 7}), // compatible structure
  '(5, 7)');
```

Conversely, in Java's nominal type system, we must explicitly declare with each class which interfaces it implements. Therefore, a class can only implement interfaces that exist at its creation time.

4.12.4 Optional properties

If a property can be omitted, we put a question mark after its name:

```
type Person = {
  name: string,
  company?: string,
};
```

In the following example, both `john` and `jane` match the type `Person`:

```
const john: Person = {
  name: 'John',
};
const jane: Person = {
  name: 'Jane',
  company: 'Massive Dynamic',
};
```

4.12.5 Methods

Object literal types can also contain methods:

```
type Point = {
  x: number,
  y: number,
  distance(other: Point): number,
};
```

As far as TypeScript's type system is concerned, method definitions and properties whose values are functions, are equivalent:

```
type HasMethodDef = {
  simpleMethod(flag: boolean): void,
};
type HasFuncProp = {
  simpleMethod: (flag: boolean) => void,
};
type _ = Assert<Equal<
  HasMethodDef,
  HasFuncProp
>>;

const objWithMethod = {
  simpleMethod(flag: boolean): void {},
};
assertType<HasMethodDef>(objWithMethod);
assertType<HasFuncProp>(objWithMethod);

const objWithOrdinaryFunction: HasMethodDef = {
  simpleMethod: function (flag: boolean): void {},
};
assertType<HasMethodDef>(objWithOrdinaryFunction);
assertType<HasFuncProp>(objWithOrdinaryFunction);

const objWithArrowFunction: HasMethodDef = {
  simpleMethod: (flag: boolean): void => {},
};
assertType<HasMethodDef>(objWithArrowFunction);
assertType<HasFuncProp>(objWithArrowFunction);
```

My recommendation is to use whichever syntax best expresses how a property should be set up.

4.13 Union types

The values that are held by a variable (one value at a time) may be members of different types. In that case, we need a *union type*. For example, in the following code, `stringOrNumber` is either of type `string` or of type `number`:

```
function getScore(stringOrNumber: string|number): number {
  if (typeof stringOrNumber === 'string'
    && /^\*{1,5}$/.test(stringOrNumber)) {
      return stringOrNumber.length;
  } else if (typeof stringOrNumber === 'number'
    && stringOrNumber >= 1 && stringOrNumber <= 5) {
    return stringOrNumber
  } else {
    throw new Error('Illegal value: ' + JSON.stringify(stringOrNumber));
  }
}

assert.equal(getScore('*****'), 5);
assert.equal(getScore(3), 3);
```

`stringOrNumber` has the type `string|number`. The result of the type expression `s|t` is the set-theoretic union of the types `s` and `t` (interpreted as sets).

4.13.1 Adding `undefined` and `null` to types

In TypeScript, the values `undefined` and `null` are not included in any type (other than the types `undefined`, `null`, `any` and `unknown`). That is common in statically type languages (with one notable exception being Java). We need union types such as `undefined|string` and `null|string` if we want to allow those values:

```
let numberOrNull: undefined|number = undefined;
numberOrNull = 123;
```

Otherwise, we get an error:

```
// @ts-expect-error: Type 'undefined' is not assignable to type 'number'.
let mustBeNumber: number = undefined;
mustBeNumber = 123;
```

Note that TypeScript does not force us to initialize immediately (as long as we don't read from the variable before initializing it):

```
let myNumber: number; // OK
myNumber = 123;
```

4.13.2 Unions of string literal types

Unions of string literals provide a quick way of defining a type with a limited set of values. For example, this is how the Node.js types define the buffer encoding that you can use (e.g.) with `fs.readFileSync()`:

```
type BufferEncoding =
  | 'ascii'
  | 'utf8'
  | 'utf-8'
  | 'utf16le'
  | 'utf-16le'
  | 'ucs2'
  | 'ucs-2'
  | 'base64'
  | 'base64url'
  | 'latin1'
  | 'binary'
  | 'hex'
;
```

It's neat that we get auto-completion for such unions (figure 4.1). We can also rename the elements of the union everywhere they are used – via the same refactoring that also changes function names.

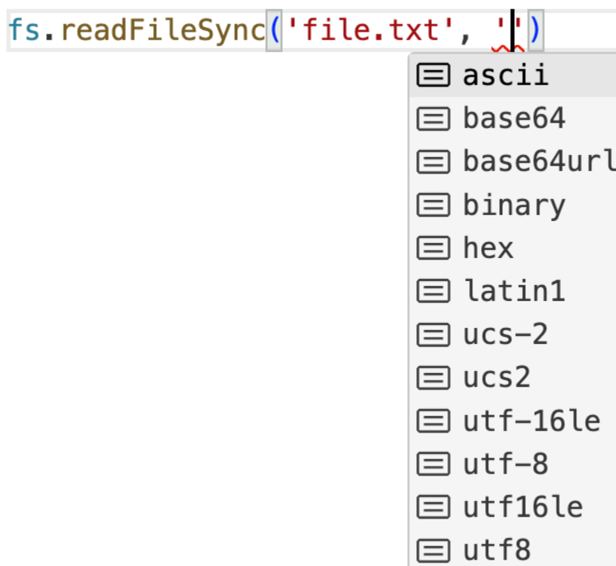

Figure 4.1: The auto-completion for `BufferEncoding` shows all elements of the union type.

4.14 Intersection types

Where a union type computes the union of two types, viewed as sets, an intersection type computes the intersection:

```
type Type1 = 'a' | 'b' | 'c';
type Type2 = 'b' | 'c' | 'd' | 'e';
type _ = Assert<Equal<
  Type1 & Type2,
  'b' | 'c'
>>;
```

The generic type Assert<B> is for comparing types

In this book, types are compared via the generic type `Assert<B>` (more information).

One key use case for intersection types is combining object types (more information).

4.15 Type guards and narrowing

Sometimes we are faced with types that are overly general. Then we need to use conditions with so-called *type guards* to make them small enough so that we can use them. That process is called *narrowing*.

In the following code, we narrow the type of `value` via the type guard `typeof`:

```
function getLength(value: string | number): number {
  assertType<string | number>(value); // (A)
  // @ts-expect-error: Property 'length' does not exist on
  // type 'string | number'.
  value.length; // (B)
  if (typeof value === 'string') {
    assertType<string>(value); // (C)
    return value.length; // (D)
  }
  assertType<number>(value); // (E)
  return String(value).length;
}
```

It's interesting to see how the type of `value` changes, due to us using `typeof` in the condition of an `if` statement:

- Initially, the type of `value` is `string | number` (line A).
  - That's why we can't access property `.length` in line B.
- Inside the true branch of the `if` statement, the type of `value` is `string` (line C).
  - Now we can access property `.length` (line D).
- Because we return from inside the true branch, TypeScript knows that `value` has type `number` in line E.

4.16 Type variables and generic types

Recall the two language levels of TypeScript:

- Values exist at the *dynamic level*.
- Types exist at the *static level*.

Similarly:

- Normal functions exist at the dynamic level, are factories for values and have parameters representing values. Parameters are declared between parentheses:

  ```
  const valueFactory = (x: number) => x; // definition
  const myValue = valueFactory(123); // use
  ```

- *Generic types* exist at the static level, are factories for types and have parameters representing types. Parameters are declared between angle brackets:

  ```
  type TypeFactory<X> = X; // definition
  type MyType = TypeFactory<string>; // use
  ```

Naming type parameters

In TypeScript, it is common to use a single uppercase character (such as `T`, `I`, and `O`) for a type parameter. However, any legal JavaScript identifier is allowed and longer names often make code easier to understand.

4.16.1 Example: a container for values

```
// Factory for types
type ValueContainer<Value> = {
  value: Value;
};

// Creating one type
type StringContainer = ValueContainer<string>;
```

`Value` is a *type variable*. One or more type variables can be introduced between angle brackets.

4.16.2 Example: a generic class

Classes can have type parameters, too:

```
class SimpleStack<Elem> {
  #data: Array<Elem> = [];
  push(x: Elem): void {
    this.#data.push(x);
  }
  pop(): Elem {
    const result = this.#data.pop();
```

```
      if (result === undefined) {
        throw new Error();
      }
      return result;
    }
    get length() {
      return this.#data.length;
    }
  }
```

Class `SimpleStack` has the type parameter `Elem`. When we instantiate the class, we also provide a value for the type parameter:

```
const stringStack = new SimpleStack<string>();
stringStack.push('first');
stringStack.push('second');
assert.equal(stringStack.length, 2);
assert.equal(stringStack.pop(), 'second');
```

4.16.3 Example: Maps

Maps are typed generically in TypeScript. For example:

```
const myMap: Map<boolean,string> = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
```

Thanks to type inference (based on the argument of `new Map()`), we can omit the type parameters:

```
const myMap = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
assertType<Map<boolean, string>>(myMap);
```

4.16.4 Functions and methods with type parameters

Function definitions can introduce type variables like this:

```
function identity<Arg>(arg: Arg): Arg {
  return arg;
}
```

We use the function as follows:

```
const num1 = identity<number>(123);
assertType<number>(num1);
```

Due to type inference, we can once again omit the type parameter:

```
const num2 = identity(123);
assertType<123>(num2);
```

The type of `num2` is the number literal type `123`.

Arrow functions with type parameters

Arrow functions can also have type parameters:

```
const identity = <Arg>(arg: Arg): Arg => arg;
```

Methods with type parameters

This is the type parameter syntax for methods:

```
const obj = {
  identity<Arg>(arg: Arg): Arg {
    return arg;
  },
};
```

A more complicated function example

```
function fillArray<T>(len: number, elem: T): T[] {
  return new Array<T>(len).fill(elem);
}
```

The type variable `T` appears four times in this code:

- It is introduced via `fillArray<T>`. Therefore, its scope is the function.
- It is used for the first time in the type annotation for the parameter `elem`.
- It is used for the second time to specify the return type of `fillArray()`.
- It is also used as a type argument for the constructor `Array()`.

We can omit the type parameter when calling `fillArray()` (line A) because TypeScript can infer `T` from the parameter `elem`:

```
const arr1 = fillArray<string>(3, '*');
assertType<string[]>(arr1);
assert.deepEqual(
  arr1, ['*', '*', '*']);

const arr2 = fillArray(3, '*'); // (A)
assertType<string[]>(arr2);
```

4.17 Conclusion: understanding the initial example

Let's use what we have learned to understand the piece of code we have seen earlier:

```
interface Array<T> {
  concat(...items: Array<T[] | T>): T[];
  reduce<U>(
```

```
    callback: (state: U, element: T, index: number) => U,
    firstState?: U
  ): U;
  // ···
}
```

This is an interface for Arrays whose elements are of type `T`:

- method `.concat()`:
  - Has zero or more parameters (defined via a rest parameter). Each of those parameters has the type `T[]|T`. That is, it is either an Array of `T` values or a single `T` value. That means that the values in `items` have the same type `T` as the values in `this` (the receiver of the method call).
  - Returns an Array whose elements also have the type `T`.
- method `.reduce()` introduces its own type variable `U`. `U` is used to express the fact that the following entities all have the same type:
  - Parameter `state` of `callback()`
  - Result of `callback()`
  - Optional parameter `firstState` of `.reduce()`
  - Result of `.reduce()`

  In addition to `state`, `callback()` has the following parameters:

  - `element`: which has the same type `T` as the Array elements
  - `index`: a number

4.18 Next steps

- Next, you'll probably want to read "How TypeScript is used: workflows, tools, etc." (§6) – which gives you a better understanding of how TypeScript is used in practice.
- Then you can move on to the rest of the book.

While using TypeScript, keep the following tip in mind.

4.18.1 Tip: Use `strict` type checking whenever you can

There are many ways in which the TypeScript compiler can be configured. One important group of options controls how strictly the compiler checks TypeScript code. My recommendation is:

- Option `strict` should always be enabled.
- There are a few additional settings that increase strictness even further: I'd start with all of them and deactivate those whose errors you don't like or don't want to deal with.

You may be tempted to use settings that produce fewer compiler errors. However, without `strict` checking, TypeScript simply doesn't work as well and will detect far fewer problems in your code.

For more information on configuring TypeScript, see "Guide to `tsconfig.json`" (§8).

Chapter 5

Notation used in this book

5.1 JavaScript level: `assert.*` . . . 53
5.2 Type level: `assertType<T>(v)` . . . 54
5.3 Type level: `Assert<B>` . . . 54
5.4 Type level: `@ts-expect-error` . . . 55
5.5 Isn't this book's notation kind of ugly? . . . 55

This chapter explains functionality that is used in the code examples to explain results and errors. We have to consider two levels:

- JavaScript level: values (e.g. returned by functions) and exceptions
- Type level: types (e.g. constructed by generic types) and compiler errors

The functions and generic types that help us, have to be imported: The import statements to do so are shown in this chapter, but omitted elsewhere in this book.

5.1 JavaScript level: `assert.*`

Expected results are checked via the following assertion functions from the Node.js module `node:assert`:

- `assert.equal()` tests equality via `===`
- `assert.deepEqual()` tests equality by deeply comparing nested objects (incl. Arrays).
- `assert.throws()` complains if the callback parameter does *not* throw an exception.

This is an example of using these assertions:

```
import assert from 'node:assert/strict';

assert.equal(
  3 + ' apples',
```

```
  '3 apples'
);

assert.deepEqual(
  [...['a', 'b'], ...['c', 'd']],
  ['a', 'b', 'c', 'd']
);

assert.throws(
  () => Object.freeze({}).prop = true,
  /^TypeError: Cannot add property prop, object is not extensible/
);
```

In the first line, the specifier of the imported module has the suffix `/strict`. That enables strict assertion mode, which uses === and not == for comparisons.

5.2 Type level: `assertType<T>(v)`

Function `assertType()` is provided by the TypeScript library `asserttt`.

The function call `assertType<T>(v)` asserts that the (dynamic) value `v` has the (static) type `T`:

```
import { assertType } from 'asserttt';

let value = 123;
assertType<number>(value);
```

5.3 Type level: `Assert<B>`

`asserttt` also provides the utility type `Assert<B>`, which asserts that the type `B` (usually an instantiated generic type) is `true`:

```
import { type Assert, type Equal, type Not } from 'asserttt';

type Pair<X> = [X, X];
type _ = [
  Assert<Equal<
    Pair<'a'>, ['a', 'a']
  >>,
  Assert<Not<Equal<
    Pair<'a'>, ['x', 'x']
  >>>,
];
```

`asserttt` has several *predicates* (generic types that construct booleans) that we can use with `Assert<>`. In the previous example, we have used:

- `Equal<T1, T2>`
- `Not<B>`

5.4 Type level: `@ts-expect-error`

In this book, `@ts-expect-error` is used to show TypeScript compiler errors:

```
// @ts-expect-error: The value 'null' cannot be used here.
const value = null.myProp;
```

How does TypeScript handle such a directive?

- If there is an error in a line after a `@ts-expect-error` comment then that error is ignored and compilation succeeds.
- If there is no error then TypeScript complains:

  ```
  Unused '@ts-expect-error' directive.
  ```

In other words: TypeScript checks that there is an error but not what error it is. All text after `@ts-expect-error` is ignored (including the colon).

To get more thorough checks, I use the tool `ts-expect-error` which checks if the suppressed error messages match the texts after `@ts-expect-error:`.

5.5 Isn't this book's notation kind of ugly?

When it comes to displaying type information for TypeScript code, there are some very pretty approaches out there – e.g. Shiki Twoslash which uses the twoslash syntax.

This book uses in-code checks (as described above) even though that doesn't look as nice. Why?

- This notation makes you think about types in terms of tests. That prepares you for computed types and for coding exercises – whose notation is similar.
- The notation makes it possible to test the code examples automatically, via the Markcheck tool for Markdown. That ensures that they don't contain errors. Twoslash only specifies which types to display; it does not check that those types are as expected.
- For printed books, HTML still isn't where I'd like it to be. Thus, I can't use Shiki Twoslash there.
- Minor downside of Shiki Twoslash: You need to run the TypeScript type checker in order to render a book. With my notation, I only need to run it when I check the code examples.

Chapter 6

How TypeScript is used: workflows, tools, etc.

6.1 TypeScript is JavaScript plus type syntax 58
6.2 Ways of running TypeScript code 58
  6.2.1 Running TypeScript directly 59
  6.2.2 Bundling TypeScript 59
  6.2.3 Transpiling TypeScript to JavaScript 59
  6.2.4 The filename extensions of locally imported TypeScript modules . . 60
6.3 Publishing a library package to the npm registry 60
  6.3.1 Essential: `.js` and `.d.ts` 61
  6.3.2 Optional: source maps 62
6.4 DefinitelyTyped: a repository with types for type-less npm packages . . . 63
6.5 Compiling TypeScript with tools other than `tsc` 63
  6.5.1 Type stripping 63
  6.5.2 Isolated declarations 64
6.6 JSR – the JavaScript registry 65
  6.6.1 Who owns JSR? 65
6.7 Editing TypeScript 65
6.8 Type-checking JavaScript files 66

Read this chapter if you are a JavaScript programmer and want to get a rough idea of what using TypeScript is like (think first step before learning more details). You'll get answers to the following questions:

- How is TypeScript code different from JavaScript code?
- How is TypeScript code run?
- How does TypeScript help during editing in an IDE?
- Etc.

This chapter focuses on how TypeScript works. If you want to know more about why it is useful, see "Sales pitch for TypeScript" (§2).

6.1 TypeScript is JavaScript plus type syntax

Let's start with a rough first description of what TypeScript is. That description is not completely accurate (there are exceptions and details that I'm omitting), but it should give you a solid first idea:

> TypeScript is JavaScript plus type syntax.

What are the purposes of these two parts?

- The JavaScript syntax is what is run and what exists at runtime: In order to run TypeScript code, the type syntax must be removed – via compilation that results in pure JavaScript. That code is executed by a JavaScript engine.
- The type syntax is only used during editing and compiling; it has no effect at runtime:
  - On one hand, it supports *type checking* which reports errors if there are inconsistencies within the types or between the types and the JavaScript values. Type checking runs during editing and during compiling.
  - On the other hand, the types improve editing via auto-completion, type hints, refactorings, etc.

Consider the following TypeScript code:

```
function add(x: number, y: number): number {
  return x + y;
}
```

If we want to run this code, we have to remove the type syntax and get JavaScript that is executed by a JavaScript engine:

```
function add(x, y) {
  return x + y;
}
```

6.2 Ways of running TypeScript code

Consider the following TypeScript project:

```
ts-app/
  tsconfig.json
  src/
    main.ts
    util.ts
    util_test.ts
  test/
    integration_test.ts
```

- `tsconfig.json` is a configuration file that tells TypeScript how to type-check and compile our code.
- The remaining files are TypeScript source code.

Let's explore the different ways in which we can run this code.

6.2.1 Running TypeScript directly

Most server-side runtimes now can run TypeScript code directly – e.g., Node.js, Deno and Bun. In other words, the following works in Node.js 23.6.0+:

```
cd ts-app/
node src/main.ts
```

6.2.2 Bundling TypeScript

When developing a web app, *bundling* is a common practice – even for pure JavaScript projects: All the JavaScript code (app code and library code) is combined into a single JavaScript file (sometimes more, but never more than a few) – which is typically loaded from an HTML file. That has several benefits:

- Prior to HTTP/2, only one file could be served per connection. But that benefit of bundling is not relevant anymore.
  - Each file the client has to request and process, still incurs a little overhead (even though no new connection is opened).
- Web servers don't have to serve many (often small) files – which helps with efficiency.
- A single large file can be compressed better than many small files.

Most bundlers support TypeScript – either directly or via plugins. That means, we run our TypeScript code via the JavaScript file `bundle.js` that was produced by a bundler:

```
ts-app/
  tsconfig.json
  src/
    main.ts
    util.ts
    util_test.ts
  test/
    integration_test.ts
  dist/
    bundle.js
```

6.2.3 Transpiling TypeScript to JavaScript

Another option is to compile out TypeScript app to JavaScript via the TypeScript compiler `tsc` and run the resulting code. Before server-side JavaScript runtimes had built-in support for TypeScript, that was the only way we could run TypeScript there.

Compiling source code to source code is also called *transpiling*. `tsconfig.json` specifies where the transpilation output is written. Let's assume we write it to the directory `dist/`:

```
ts-app/
  tsconfig.json
  src/
    main.ts
    util.ts
    util_test.ts
  test/
    integration_test.ts
  dist/
    src/
      main.js
      util.js
      util_test.js
    test/
      integration_test.js
```

6.2.4 The filename extensions of locally imported TypeScript modules

When it comes to filename extensions of locally imported TypeScript modules, we must distinguish between code that is transpiled and code that is run directly.

By default, TypeScript does not change the specifiers of imported modules. Therefore, code that is transpiled must look like this (we import `util.js`, from JavaScript code):

```
// main.ts
import {helperFunc} from './util.js';
```

However, such code does not work if we run it directly. There, we must write (we import `util.ts` from TypeScript code):

```
// main.ts
import {helperFunc} from './util.ts';
```

We can also tell TypeScript to change the filename extensions of local imports from `.ts` to `.js` (more information). Then the previous code can also be transpiled.

6.3 Publishing a library package to the npm registry

The npm registry is still the most popular means of publishing packages. Even though Node.js runs TypeScript code, packages must be deployed as JavaScript code. That enables JavaScript code to use library packages written in TypeScript. However, we additionally want to support TypeScript features. Therefore, a single library file `lib.ts` is often deployed as five files (four of which are compiled by TypeScript from `lib.ts`):

- Essential:
  - `lib.js`: the JavaScript part of `lib.ts`
  - `lib.d.ts`: the type part of `lib.ts` (a *declaration file*)
- Optional: source maps. They map source code locations of compilation output to `lib.ts`.
  - `lib.js.map`: source map for `lib.js`
  - `lib.d.ts.map`: source map for `lib.d.ts`

  - `lib.ts`: the target of the previous two source maps

(More on what all of that means in a second.)

As an example, consider the following library package:

```
ts-lib/
  package.json
  tsconfig.json
  src/
    lib.ts
  dist/
    lib.js
    lib.js.map
    lib.d.ts
    lib.d.ts.map
```

- `package.json` is npm's description of our library package. Some of its data, such as the so-called *package exports*, are also used by TypeScript – e.g. to look up type information when someone imports from our package.
- Every file in `dist/` was generated by TypeScript. While it is uploaded to the npm registry, it is usually not added to version control systems because it can easily be regenerated.
- Only `tsconfig.json` is not uploaded to the npm registry.

6.3.1 Essential: `.js` and `.d.ts`

It's interesting to see the combined JavaScript plus types in `.lib.ts` be split into `lib.js` with only JavaScript and `lib.d.ts` with only types. Why do that? It enables library packages to be used by either JavaScript code or TypeScript code:

- JavaScript code can ignore `.d.ts` files.
- TypeScript uses them for type checking, auto-completion, refactorings, etc.

Actually, behind the scenes, many editors (e.g. Visual Studio Code) use a kind of lightweight TypeScript mode when editing JavaScript code so that we also get simple type checking and code completion there.

This is the TypeScript input `lib.ts`

```
/** Add two numbers. */
export function add(x: number, y: number): number {
  return x + y; // numeric addition
}
```

It is split into `lib.js` on one hand:

```
/** Add two numbers. */
export function add(x, y) {
    return x + y; // numeric addition
}
//# sourceMappingURL=lib.js.map
```

And `lib.d.ts` on the other hand:

```
/** Add two numbers. */
export declare function add(x: number, y: number): number;
//# sourceMappingURL=lib.d.ts.map
```

Notes:

- Both files point to their source maps.
- By default, both files contain comments (but we can tell TypeScript not to include them):
  - `lib.js` has all comments so that the code is easier to read.
  - `lib.d.ts` only has JSDoc comments (`/** */`) because they are used by many IDEs to display inline documentation.

6.3.2 Optional: source maps

If we compile a file `I` to a file `O` then a source map for `O` maps source code locations in `O` to source code locations in `I`. That means we can work with `O` but display information from `I` – e.g.:

- `lib.js.map`: maps `lib.js` locations to `lib.ts` locations and gives us debugging and stack traces for the latter when we run the former.
- `lib.d.ts.map`: maps `lib.d.ts` lines to `lib.ts` lines. It enables "go to definition" for imports from `lib.ts` to take us to that file.

All source-map-related functionality except stack traces require access to the original TypeScript source code. That's why it makes sense to include `lib.ts` if there are source maps.

This is what `lib.js.map` looks like:

```
{
  "version": 3,
  "file": "lib.js",
  "sourceRoot": "",
  "sources": [
    "../../src/lib.ts"
  ],
  "names": [],
  "mappings": "AAAA,uBAAuB;AACvB,MAAM,UAAU,···"
}
```

This is what `lib.d.ts.map` looks like:

```
{
  "version": 3,
  "file": "lib.d.ts",
  "sourceRoot": "",
  "sources": [
    "../../src/lib.ts"
  ],
  "names": [],
```

```
  "mappings": "AAAA,uBAAuB;AACvB,wBAAgB,GAAG,···"
}
```

In both cases, the actual content of `"mappings"` was abbreviated. And in the actual output of `tsc`, the JSON is always squeezed into a single line.

6.4 DefinitelyTyped: a repository with types for type-less npm packages

These days, many npm packages come with TypeScript types. However, not all of them do. In that case, DefinitelyTyped may help: If it supports a type-less package `pkg` then we can additionally install a package `@types/pkg` with types for `pkg`.

One important DefinitelyTyped package for Node.js is `@types/node` with types for all of its APIs. If you develop TypeScript on Node.js, you will usually have this package as a development dependency.

6.5 Compiling TypeScript with tools other than `tsc`

Let's recap all the tasks performed by `tsc` (we'll ignore source maps in this section):

1. It compiles TypeScript files to JavaScript files.
2. It compiles TypeScript files to type declaration files.
3. It type-checks TypeScript files.

##3 is so complex that only `tsc` can do it. However, for both #1 and #2, there are slightly simpler subsets of TypeScript where compilation does not involve much more than syntactic processing. That means that we can use external, faster tools for #1 and #2.

There are even `tsconfig.json` settings to warn us if we don't stay within those subsets of TypeScript (more information). Doing that is not much of a sacrifice in practice.

6.5.1 Type stripping

Type stripping is a simple and fast way of compiling TypeScript to JavaScript. It's what Node.js uses when it runs TypeScript. Type stripping is fast because it only supports a subset of TypeScript where two things are possible:

1. Type syntax can be detected and removed by only parsing the syntax – without performing additional semantic analyses.

2. No non-type language features are transpiled. In other words: Removing the type syntax is enough to produce JavaScript.

#2 means that there are several TypeScript features that we can't use – e.g., enums and JSX (HTML-like syntax inside TypeScript, as used, e.g., by React).

One considerable benefit of type stripping is that it does not need any configuration (via `tsconfig.json` or other means) because it's so simple. That makes platforms that use it more stable w.r.t. changes made to TypeScript.

Type stripping technique: replacing types with spaces

One clever technique for type stripping was pioneered by the `ts-blank-space` tool (by Ashley Claymore for Bloomberg): Instead of simply removing the type syntax, it replaces it with spaces. That means that source code positions in the output don't change. Therefore, any positions that show up (e.g.) in stack traces still work for the input and there is less of a need for source maps: You still need them for debugging and going to definitions but JavaScript generated by type stripping is relatively close to the original TypeScript and you are often OK even then.

For example - input (TypeScript):

```
function add(x: number, y: number): number {
  return x + y;
}
```

Output (JavaScript):

```
function add(x        , y        )         {
  return x + y;
}
```

If you want to explore further, you can check out the ts-blank-space playground.

6.5.2 Isolated declarations

"Isolated declaration" is a style of writing TypeScript that makes it easier for external tools to generate declaration files. It mainly means we have to add type annotations in locations where the TypeScript compiler `tsc` does not need them – thanks to its ability to automatically derive types (so-called *type inference*). However, external tools are simpler and faster if they don't need that ability. Additionally, they don't have to visit and analyze external files if the type information is provided locally.

The following example shows how the isolated declaration style changes code:

```
// OK: return type stated explicitly
export function f1(): string {
  return 123..toString();
}
// Error: return type requires inference
export function f2() {
  return 123..toString();
}
// OK: return type trivial to determine
export function f3() {
  return 123;
}
```

Note that isolated declarations only affect constructs that are exported. Module-internal code does not show up in declaration files.

6.6 JSR – the JavaScript registry

The JavaScript registry JSR is an alternative to npm and the npm registry for publishing packages. It works as follows:

- For TypeScript packages, you only upload `.ts` files.
- How to install a TypeScript package depends on the platform:
  - On JavaScript platforms where TypeScript-only library packages are supported, JSR only installs TypeScript.
  - On all other platforms, JSR automatically generates `.js` files and `.d.ts` files and installs those, along with the `.ts` files. To make automatic generation possible, the TypeScript code must follow a set of rules called "no slow types" – which is similar to isolated declarations.

In contrast, with the npm registry, your TypeScript library package is only usable on Node.js if you upload `.js` files and `.d.ts` files.

JSR also provides several features that npm doesn't such as automatic generation of documentation. See "Why JSR?" in the official documentation for more information.

6.6.1 Who owns JSR?

Quoting the official documentation page "Governance":

> JSR is not owned by any one person or organization. It is a community-driven project that is open to all, built for the entire JavaScript ecosystem.
>
> JSR is currently operated by the Deno company. We are currently working on establishing a governance board to oversee the project, which will then work on moving the project to a foundation.

6.7 Editing TypeScript

Two popular IDEs for JavaScript are:

- *Visual Studio Code* (free)
- *WebStorm* (free for non-commercial use)

The observations in this section are about Visual Studio Code, but may apply to other IDEs, too.

With Visual Studio Code, we get two different ways of type checking:

- Any file that is currently open is automatically type-checked within Visual Studio Code. It order to provide that functionality, it comes with its own installation of TypeScript.

- If we want to type-check all of a code base, we must invoke the TypeScript compiler `tsc`. We can do that via Visual Studio Code's *tasks* – a built-in way of invoking external tools (for type checking, compiling, bundling, etc.). The official documentation has more information on tasks.

6.8 Type-checking JavaScript files

Optionally, TypeScript can also type-check JavaScript files. Obviously that will only give us limited results. However, to help TypeScript, we can add type information via JSDoc comments – e.g.:

```
/**
 * @param {number} x - The first operand
 * @param {number} y - The second operand
 * @returns {number} The sum of both operands
 */
function add(x, y) {
  return x + y;
}
```

If we do that, we are still writing TypeScript, just with a different syntax.

Benefits of this approach:

- No need for a build step to run the code – even on platforms (such as browsers) that don't support TypeScript.
  - We can also generate `.d.ts` files from `.js` files with JSDoc comments. That is an extra build step, though. How to do that is explained in the TypeScript Handbook.
- It enables us to make a JavaScript code base more type-safe – in small incremental steps.

Downside of this approach:

- The syntax becomes less pleasant to use.

To explain the downside – consider how we define an interface in TypeScript:

```
interface Point {
  x: number;
  y: number;
  /** optional property */
  z?: number;
}
```

Doing that via a JSDoc comment looks like this:

```
/**
 * @typedef Point
 * @prop {number} x
 * @prop {number} y
 * @prop {number} [z] optional property
 */
```

More information in the TypeScript Handbook:

- "Type Checking JavaScript Files"
- "Creating `.d.ts` Files from `.js` files"
- "JSDoc Reference"

Chapter 7

Trying out TypeScript without installing it

7.1 The TypeScript Playground 67
7.2 A simple TypeScript playground via `node --watch` 68
  7.2.1 More configuration: `tsconfig.json` and `package.json` 68
  7.2.2 Further reading 68
7.3 Running copied TypeScript code via Node.js 68

This chapter gives tips for quickly trying out TypeScript.

7.1 The TypeScript Playground

The *TypeScript Playground* is an online editor for TypeScript code. Features include:

- Supports full IDE-style editing: auto-completion, etc.
- Displays static type errors.
- Shows the result of compiling TypeScript code to JavaScript and declarations (`.d.ts`).
- Can run the JavaScript output in the browser.

The Playground is very useful for quick experiments and demos. It can save both TypeScript code snippets and compiler settings into URLs, which is great for sharing such snippets with others. This is an example of such a URL:

```
https://www.typescriptlang.org/play/?#code/«base64»
```

Many social media services limit the characters per post, but not the characters per URL. Therefore, we can use Playground URLs to share code that wouldn't fit into a post.

7.2 A simple TypeScript playground via `node --watch`

This is the basic approach:

- We create a file `playground.mts`
  - The filename extension is `.mts` so that we don't need a `package.json` file to tell Node.js that `.ts` means ESM module.
- We run the following command in a terminal:

  ```
  node --watch ./playground.mts
  ```

- We edit `playground.mts`. Whenever we save it, Node.js re-runs it and shows its output.

Note that Node.js does not type-check the code. But the type checking we get in TypeScript editors should be enough in this case (since we are only working with a single file).

7.2.1 More configuration: `tsconfig.json` and `package.json`

For more sophisticated experiments, we may need two additional files:

- A `tsconfig.json` with our preferred settings
- A `package.json` with `"type":"module"` so that we can use the filename extension `.ts`.

I have created the GitHub repository `nodejs-type-stripping` where both are already set up correctly.

7.2.2 Further reading

- The relevant Node.js command line options are:
  - `--watch` Watches a file and its imports for changes and re-runs the file whenever that happens.
  - `--watch-path` Overrides the default of watching the imports and tells Node.js which files to watch instead.
  - `--watch-preserve-output` Disables the clearing of the console before the file is re-run.
- "Running TypeScript directly (without generating JS files)" (§8.6.2)

7.3 Running copied TypeScript code via Node.js

For simple experiments, it can be enough to simply copy TypeScript code and run it via Node.js:

```
«paste-command» | node --input-type=module-typescript
```

«paste-command» depends on your operating system:

- MacOS: `pbpaste`
- Windows PowerShell: `Get-Clipboard`

- Linux: There are many options. wl-clipboard worked well for me on Ubuntu, where I installed it via the App Center (snap).

On macOS, I added the following line to my `.zprofile`:

```
alias pbts='pbpaste | node --input-type=module-typescript'
```

Part III

Setting up TypeScript

Chapter 8

Guide to `tsconfig.json`

8.1 Features not covered by this chapter 74
8.2 Extending base files via `extends` 75
8.3 Where are the input files? 75
8.4 What is the output? 75
 8.4.1 Where are the output files written? 75
 8.4.2 Emitting source maps 78
 8.4.3 Emitting `.d.ts` files (e.g. for libraries) 78
 8.4.4 Fine-tuning emitted files 78
8.5 Language and platform features 79
 8.5.1 `target` 79
 8.5.2 `lib` 79
 8.5.3 `skipLibCheck` 80
 8.5.4 Types for the built-in Node.js APIs 80
8.6 Module system 80
 8.6.1 How does TypeScript look for imported modules? 80
 8.6.2 Running TypeScript directly (without generating JS files) 82
 8.6.3 Importing JSON 82
 8.6.4 Importing other non-TypeScript artifacts 83
8.7 Type checking 83
 8.7.1 `strict` 83
 8.7.2 `exactOptionalPropertyTypes` 84
 8.7.3 `noFallthroughCasesInSwitch` 85
 8.7.4 `noImplicitOverride` 86
 8.7.5 `noImplicitReturns` 86
 8.7.6 `noPropertyAccessFromIndexSignature` 86
 8.7.7 `noUncheckedIndexedAccess` 86
 8.7.8 Type checking options that have good defaults 87
8.8 Compiling TypeScript with tools other than tsc 87
 8.8.1 Using tsc only for type checking 88

8.8.2 Generating .js files via type stripping: erasableSyntaxOnly and verbatimModuleSyntax . . . 88
8.8.3 erasableSyntaxOnly: no transpiled language features . . . 88
8.8.4 verbatimModuleSyntax: enforcing type in imports and exports . . . 89
8.8.5 isolatedDeclarations: generating .d.ts files more efficiently . . . 90
8.9 Importing CommonJS from ESM . . . 92
8.9.1 allowSyntheticDefaultImports: type-checking default imports of CommonJS modules . . . 92
8.9.2 esModuleInterop: better compilation of TypeScript to CommonJS code 92
8.10 One more option with a good default . . . 93
8.11 Visual Studio Code . . . 93
8.12 Summary: Assemble your tsconfig.json by answering four questions . . . 93
8.12.1 Do you want to transpile new JavaScript to older JavaScript? . . . 95
8.12.2 Should TypeScript only allow JavaScript features at the non-type level? 95
8.12.3 Which filename extension do you want to use in local imports? . . . 95
8.12.4 What files should tsc emit? . . . 95
8.13 Further reading . . . 96
8.13.1 tsconfig.json recommendations by other people . . . 96
8.13.2 Sources of this chapter . . . 96

Version: TypeScript 5.8

This chapter covers `tsconfig.json` as supported by TypeScript 5.8.

This chapter documents all common options of the TypeScript configuration file `tsconfig.json`:

- This knowledge will enable you to understand and simplify your `tsconfig.json`.
- If you don't have the time to read the chapter, you can jump to the summary at the end where I show a starter `tsconfig.json` file with all settings – along with four questions to determine which settings you can delete.
- I also link to the `tsconfig.json` recommendations by several well-known TypeScript programmers. (I went through them when I researched this chapter.)

8.1 Features not covered by this chapter

This chapter only describes how to set up projects whose local modules are all ESM. It does give tips for importing CommonJS, though.

Not explained here:

- Importing and type-checking plain JavaScript in your code base, namely the options `allowJs` and `checkJs`.
- How to set up JSX. See "JSX" in the TypeScript Handbook.

- "Projects" (useful for monorepos): option `composite` etc. For more information on this topic, see:
  - Chapter "Project References" in the TypeScript Handbook
  - My blog post "Simple monorepos via npm workspaces and TypeScript project references"

8.2 Extending base files via extends

This option lets us refer to an existing `tsconfig.json` via a module specifier (as if we imported a JSON file). That file becomes the *base* that our tsconfig *extends*. That means that our tsconfig has all the option of the base, but can override any of them and can add options not mentioned in the base.

The GitHub repository `tsconfig/bases` lists bases that are available under the npm namespace `@tsconfig` and can be used like this (after they were installed locally via npm):

```
{
  "extends": "@tsconfig/node-lts/tsconfig.json",
}
```

Alas, none of these files suit my needs. But they can serve as an inspiration for your tsconfig.

8.3 Where are the input files?

```
{
  "include": ["src/**/*"],
}
```

On one hand, we have to tell TypeScript what the input files are. These are the available options:

- `files`: an exhaustive array of all input files
- `include`: Specifies the input files via an array of patterns with wildcards that are interpreted as relative to `tsconfig.json`.
- `exlude`: Specifies which files should be excluded from the `include` set of files – via an array of patterns.

8.4 What is the output?

8.4.1 Where are the output files written?

```
"compilerOptions": {
  "rootDir": "src",
  "outDir": "dist",
}
```

How TypeScript determines where to write an output file:

- It takes the input path (relative to `tsconfig.json`),

- removes the prefix specified by `rootDir` and
- "appends" the result to `outDir`.

As an example, consider the following `tsconfig.json`:

```
{
  "include": ["src/**/*"],
  "compilerOptions": {
    // Specify explicitly (don't derive from source file paths):
    "rootDir": "src",
    "outDir": "dist",
    // ···
  }
}
```

Consequences of these settings:

- Input: `src/util.ts`
  - Output: `dist/util.js`
- Input: `src/test/integration_test.ts`
  - Output: `dist/test/integration_test.js`

Putting `src/` and `test/` next to each other

I like the idea of having a separate directory `test/` that is a sibling of `src/`. However then the output files in `dist/` are more deeply nested inside the project's directory than the input files in `src/` and `test/`. That means that we can't access files such as `package.json` via relative module specifiers.

`tsconfig.json`:

```
{
  "include": ["src/**/*", "test/**/*"],
  "compilerOptions": {
    "rootDir": ".",
    "outDir": "dist",
    // ···
  }
}
```

Consequences of these settings:

- Input: `src/util.ts`
  - Output: `dist/src/util.js`
- Input: `test/integration_test.ts`
  - Output: `dist/test/integration_test.js`

Default values of `rootDir`

The default value of `rootDir` depends on the input file paths. I find that too unpredictable and always specify it explicitly. It is the longest common prefix of the input file paths.

Example 1: Default value is `'src'` (relative to the project directory)

`tsconfig.json`:

```
{
  "include": ["src/**/*"],
}
```

Files:

```
/tmp/my-proj/
  tsconfig.json
  src/
    main.ts
    test/
      test.ts
  dist/
    main.js
    test/
      test.js
```

Example 2: Default value is `'src/core/cli'`

`tsconfig.json`:

```
{
  "include": ["src/**/*"],
}
```

Files:

```
/tmp/my-proj/
  tsconfig.json
  src/
    core/
      cli/
        main.ts
        test/
          test.ts
  dist/
    main.js
    test/
      test.js
```

Example 3:

`tsconfig.json`: Default value is `'.'`

```
{
  "include": ["src/**/*", "test/**/*"],
}
```

Files:

```
/tmp/my-proj/
  tsconfig.json
```

```
src/
  main.ts
test/
  test.ts
dist/
  src/
    main.js
  test/
    test.js
```

8.4.2 Emitting source maps

```
"compilerOptions": {
  "sourceMap": true,
}
```

`sourceMap` produces source map files that point from the transpiled JavaScript to the original TypeScript. That helps with debugging and is usually a good idea.

8.4.3 Emitting .d.ts files (e.g. for libraries)

If we want TypeScript code to consume our transpiled TypeScript code, we usually should include `.d.ts` files:

```
"compilerOptions": {
  "declaration": true,
  "declarationMap": true, // enables importers to jump to source
}
```

Optionally, we can include the TypeScript source code in our npm package and activate `declarationMap`. Then importers can, e.g., click on types or go to the definition of a value and their editor will send them to the original source code.

Option declarationDir

By default, each `.d.ts` file is put next to its `.js` file. If you want to change that, you can use option `declarationDir`.

8.4.4 Fine-tuning emitted files

```
"compilerOptions": {
  "newLine": "lf",
  "removeComments": false,
}
```

The values shown above are the defaults.

- `newLine` configures the line endings for emitted files. Allowed values are:
  - `"lf"`: ”n” (Unix)
  - `"crlf"`: ”rn” (Windows)

- `removeComments`: If active, all comments in TypeScript files are omitted in transpiled JavaScript files. I'm weakly in favor of sticking with the default and not removing comments:
  - It helps with reading transpiled JavaScript – especially if the TypeScript source code isn't included.
  - Bundlers remove comments.
  - On Node.js, the added burden doesn't matter much.

8.5 Language and platform features

```
"compilerOptions": {
  "target": "ESNext", // sets up "lib" accordingly
  "skipLibCheck": true,
}
```

8.5.1 `target`

`target` determines which newer JavaScript syntax is transpiled to older syntax. For example, if the target is `"ES5"` then an arrow function `() => {}` is transpiled to a function expression `function () {}`. Values can be:

- `"ESNext"`
- `"ES5"`
- `"ES6"`
- `"ES2015"` (same as `"ES6"`)
- `"ES2016"`
- Etc.

`"ESNext"` means that nothing is ever transpiled. I find that setting easiest to deal with. It's also the best setting if you don't use `tsc` and use type stripping (which never transpiles anything either).

How to pick a good `"ES20YY"` target

If we want to transpile, we have to pick an ECMAScript version that works for our target platforms. There are two tables that provide good overviews:

- For browsers: `compat-table.github.io`
- For Node.js: `node.green`

Additionally, the official tsconfig bases all provide values for `target`.

8.5.2 `lib`

`lib` determines which types for built-in APIs are available – e.g. `Math` or methods of built-in types:

- There are categories such as `"ES2024"` and `"DOM"` and subcategories such as `"DOM.Iterable"` and `"ES2024.Promise"`.

- Which values are available? We can look them up here:

  - Auto-completion (e.g. in Visual Studio Code)
  - TypeScript documentation
  - TypeScript source code repository
- The values are case-insensitive: Visual Studio Code's autocompletion suggestions contain many capital letters; the filenames contain none. `lib` values can be written either way.

When does TypeScript support a given API? It must be "available un-prefixed / flagged in at least 2 browser *engines* (i.e. not just 2 chromium browsers)" (source).

Setting up `lib` via `target`

`target` determines the default value of `lib`: If the latter is omitted and `target` is `"ES20YY"` then `"ES20YY.Full"` is used. However, that is not a value we can use ourselves. If we want to replicate what removing `lib` does, we have to enumerate the contents of (e.g.) `es2024.full.d.ts` in the TypeScript source code repository ourselves:

```
/// <reference lib="es2024" />
/// <reference lib="dom" />
/// <reference lib="webworker.importscripts" />
/// <reference lib="scripthost" />
/// <reference lib="dom.iterable" />
/// <reference lib="dom.asynciterable" />
```

In this file, we can observe an interesting phenomenon:

- Category `"ES20YY"` usually includes all of its subcategories.
- Category `"DOM"` doesn't – e.g., subcategory `"DOM.Iterable"` is not yet part of it.

Among other things, `"DOM.Iterable"` enables iteration over NodeLists – e.g.:

```
for (const x of document.querySelectorAll('div')) {}
```

8.5.3 `skipLibCheck`

- `skipLibCheck:false` – By default, TypeScript type-checks all `.d.ts` files. This is normally not necessary but helps when a project contains hand-written `.d.ts` files.
- `skipLibCheck:true` – If we switch it off, then TypeScript will only type-check library functionality we use in our code. That saves time – which is why I went with `true`.

8.5.4 Types for the built-in Node.js APIs

The types for the Node.js APIs must be installed via an npm package:

```
npm install @types/node
```

8.6 Module system

8.6.1 How does TypeScript look for imported modules?

These options affect how TypeScript looks for imported modules:

```
"compilerOptions": {
  "module": "NodeNext",
  "noUncheckedSideEffectImports": true,
}
```

Option `module`

With this option, we specify systems for handling modules. If we set it up correctly, we also take care of the related option `moduleResolution`, for which it provides good defaults. The TypeScript documentation recommends either of the following two values:

- Node.js: `"NodeNext"` supports both CommonJS and the latest ESM features.
  - Implies `"moduleResolution": "NodeNext"`
  - Downside of `"NodeNext"`: It's a moving target. But generally, functionality is only added.
  - Upside of `"NodeNext"`: It supports a good mix of features – for example (source):
    * `"Node16"` does not support import attributes (which are needed for importing JSON files).
    * `"Node18"` and `"Node20"` support the outdated import assertions.
    * `require(esm)` (which is only relevant for CommonJS code, not for ESM code) is only supported by `"Node20"` and `"NodeNext"`.
- Bundlers: `"Preserve"` supports both CommonJS and the latest ESM features. It matches what most bundlers do.
  - Implies `"moduleResolution": "bundler"`

Given that bundlers mostly mimic what Node.js does, I'm always using `"NodeNext"` and haven't encountered any issues.

Note that in both cases, TypeScript forces us to mention the complete names of local modules we import. We can't omit filename extensions as was frequent practice when Node.js was only compiled to CommonJS. The new approach mirrors how pure-JavaScript ESM works.

`module:NodeNext` implies `target:ESNext` but in this case, I prefer to manually set up `target` because `module` and `target` are not as closely related as `module` and `moduleResolution`. Furthermore, `module:Bundler` does not imply anything.

Option `noUncheckedSideEffectImports`

By default, TypeScript does not complain if an empty import does not exist. The reason for this behavior is that this is a pattern supported by some bundlers to associate non-TypeScript artifacts with modules. And TypeScript only sees TypeScript files. This is what such an import looks like:

```
import './component-styles.css';
```

Interestingly, TypeScript normally is also OK with emptily imported TypeScript files that don't exist. It only complains if we import something from a non-existent file.

```
import './does-not-exist.js'; // no error!
```

Setting `noUncheckedSideEffectImports` to `true` changes that. I'm explaining an alternative for importing non-TypeScript artifacts later.

8.6.2 Running TypeScript directly (without generating JS files)

```
"compilerOptions": {
  "allowImportingTsExtensions": true,
  // Only needed if compiling to JavaScript:
  "rewriteRelativeImportExtensions": true,
}
```

Most non-browser JavaScript platforms now can run TypeScript code directly, without transpiling it.

This mainly affects what filename extension we use when we import a local module. Traditionally, TypeScript does not change module specifiers and we have to use the filename extension `.js` in ESM modules (which is what works in the JavaScript that our TypeScript is compiled to):

```
import {someFunc} from './lib/utilities.js';
```

If we run TypeScript directly, that import statement looks like this:

```
import {someFunc} from './lib/utilities.ts';
```

This is enabled via the following settings:

- `allowImportingTsExtensions`: If this option is active, TypeScript won't complain if we use the filename extension `.ts`.
- `rewriteRelativeImportExtensions`: With this option, we can also transpile TypeScript code that is meant to be run directly. By default, TypeScript does not change the module specifiers of imports. This option comes with a few caveats:
  - Only relative paths are rewritten.
  - They are rewritten "naively" – without taking the options `baseUrl` and `paths` into consideration (which are beyond the scope of this chapter).
  - Paths that are routed via the `"exports"` and `"imports"` properties in `package.json` don't look like relative paths and are therefore not rewritten either.

Related option:

- If you want to use tsc only for type checking, then take a look at the `noEmit` option.

Node's built-in support for TypeScript

Node.js now supports TypeScript via *type stripping*:

- More information on type stripping and option `erasableSyntaxOnly` that helps with it
- Node's official documentation on its TypeScript support

8.6.3 Importing JSON

```
"compilerOptions": {
  "resolveJsonModule": true,
}
```

The option `resolveJsonModule` enables us to import JSON files:

```
import data from './data.json' with {type: 'json'};
console.log(data.version);
```

8.6.4 Importing other non-TypeScript artifacts

Whenever we import a file `basename.ext` whose extension `ext` TypeScript doesn't know, it looks for a file `basename.d.ext.ts`. If it can't find it, it raises an error. The TypeScript documentation has a good example of what such a file can look like.

There are two ways in which we can prevent TypeScript from raising errors for unknown imports.

First, we can use option `allowArbitraryExtensions` to prevent any kind of error reporting in this case.

Second, we can create an *ambient module declaration* with a wildcard specifier – a `.d.ts` file that has to be somewhere among the files that TypeScript is aware of. The following example suppresses errors for all imports with the filename extension `.css`:

```
// ./src/globals.d.ts
declare module "*.css" {}
```

8.7 Type checking

```
"compilerOptions": {
  "strict": true,
  "exactOptionalPropertyTypes": true,
  "noFallthroughCasesInSwitch": true,
  "noImplicitOverride": true,
  "noImplicitReturns": true,
  "noPropertyAccessFromIndexSignature": true,
  "noUncheckedIndexedAccess": true,
}
```

`strict` is a must, in my opinion. With the remaining settings, you have to decide for yourself if you want the additional strictness for your code. You can start by adding all of them and see which ones cause too much trouble for your taste.

8.7.1 `strict`

The compiler setting `strict` provides an important minimal setting for type checking. In principle, this setting would default to `true` but backward compatibility makes that impossible.

The compiler option `strict` in a nutshell

`strict` basically means: Type-check as much as possible, as correctly as possible.

`strict` activates the following settings (which won't be mentioned again in this chapter):

- `alwaysStrict`: always emit `"use strict"` in script files. That's a legacy JavaScript feature that's not needed in ECMAScript modules.
- `noImplicitAny`: If `true`, we can omit types in some locations (mainly parameter definitions) and TypeScript will (implicitly) infer the type `any`. If `false`, we must provide explicit type annotations – which can use the type `any` (explicitly). For more information, see "The compiler option `noImplicitAny`" (§14.2.3).
- `noImplicitThis`: If we use `this` in ordinary functions, we must explicitly declare its type.
- `strictBindCallApply`: If `true`, TypeScript will check that we pass correct arguments to `.call()`, `.apply()` and `.bind()`. If `false`, we can pass any arguments to those methods.
- `strictBuiltinIteratorReturn`: If active, built-in iterators have the `TReturn` type `undefined` (instead of `any`).
- `strictFunctionTypes`: If `true`, compatibility between function types is handled more correctly.
- `strictNullChecks`: If `true`, the values `undefined` and `null` are not elements of normal types `T`. If we want to accept them, we have to use the type `undefined | T` or `null | T`, respectively.
- `strictPropertyInitialization`: If `true`, TypeScript warns us if we don't initialize a class instance property in the constructor. For more information, see "Strict property initialization" (§21.4.1).
- `useUnknownInCatchVariables`: If `true`, TypeScript gives `catch` variables without type annotations the type `unknown` (instead of `any`).

8.7.2 `exactOptionalPropertyTypes`

If `true` then `.colorTheme` can only be omitted and not be set to `undefined`:

```
interface Settings {
  // Absent property means "system"
  colorTheme?: 'dark' | 'light';
}
const obj1: Settings = {}; // allowed
// @ts-expect-error: Type '{ colorTheme: undefined; }' is not
// assignable to type 'Settings' with
// 'exactOptionalPropertyTypes: true'. Consider adding 'undefined'
// to the types of the target's properties.
const obj2: Settings = { colorTheme: undefined };
```

This option also prevents optional tuple elements being `undefined` (vs. missing):

```
const tuple1: [number, string?] = [1];
const tuple2: [number, string?] = [1, 'hello'];
// @ts-expect-error: Type '[number, undefined]' is not assignable to
// type '[number, string?]'.
const tuple3: [number, string?] = [1, undefined];
```

`exactOptionalPropertyTypes` prevents useful patterns

I'm ambivalent about this option: On one hand, enabling it prevents useful patterns such as:

```
type Obj = {
  num?: number,
};
function createObj(num?: number): Obj {
  // @ts-expect-error: Type '{ num: number | undefined; }' is not
  // assignable to type 'Obj' with
  // 'exactOptionalPropertyTypes: true'.
  return { num };
}
```

`exactOptionalPropertyTypes` produces better types: spreading

On the other hand, it does better reflect how JavaScript works – e.g., spreading distinguishes missing properties and properties whose values are `undefined`:

```
const optionDefaults: { a: number } = { a: 1 };
// This assignment is an error with `exactOptionalPropertyTypes`
const options: { a?: number } = { a: undefined }; // (A)

const result = { ...optionDefaults, ...options };
assertType<
  { a: number }
>(result);
assert.deepEqual(
  result, { a: undefined }
);
```

If we had assigned an empty object in line A then the value of `result` would be `{a:1}` and match its type.

`Object.assign()` works similarly to spreading.

`exactOptionalPropertyTypes` produces better types: `in` operator

```
function f(obj: {prop?: number}): void {
  if ('prop' in obj) {
    // Without `exactOptionalPropertyTypes`, the type would be:
    // number | undefined
    assertType<number>(obj.prop);
  }
}
```

8.7.3 `noFallthroughCasesInSwitch`

If `true`, non-empty `switch` cases must end with `break`, `return` or `throw`.

8.7.4 `noImplicitOverride`

If `true` then methods that override superclass methods must have the `override` modifier.

8.7.5 `noImplicitReturns`

If `true` then an "implicit return" (the function or method ending) is only allowed if the return type is `void`.

8.7.6 `noPropertyAccessFromIndexSignature`

If `true` then for types such as the following one, we cannot use the dot notation for unknown properties, only for known ones:

```
interface ObjectWithId {
  id: string,
  [key: string]: string;
}
function f(obj: ObjectWithId) {
  const value1 = obj.id; // allowed
  const value2 = obj['unknownProp']; // allowed
  // @ts-expect-error: Property 'unknownProp' comes from an index
  // signature, so it must be accessed with ['unknownProp'].
  const value3 = obj.unknownProp;
}
```

8.7.7 `noUncheckedIndexedAccess`

`noUncheckedIndexedAccess` and objects

If `noUncheckedIndexedAccess` is `true` then the type of an unknown property is the union of `undefined` and the type of the index signature:

```
interface ObjectWithId {
  id: string,
  [key: string]: string;
}
function f(obj: ObjectWithId): void {
  assertType<string>(obj.id);
  assertType<undefined | string>(obj['unknownProp']);
}
```

`noUncheckedIndexedAccess` does the same for `Record` (which is a mapped type):

```
function f(obj: Record<string, number>): void {
  // Without `noUncheckedIndexedAccess`, this type would be:
  // number
  assertType<undefined | number>(obj['hello']);
}
```

`noUncheckedIndexedAccess` and Arrays

Option `noUncheckedIndexedAccess` also affects how Arrays are handled:

```
const arr = ['a', 'b'];
const elem = arr[0];
// Without `noUncheckedIndexedAccess`, this type would be:
// string
assertType<undefined | string>(elem);
```

One common pattern for Arrays is to check the length before accessing an element. However, that pattern becomes inconvenient with `noUncheckedIndexedAccess`:

```
function logElemAt0(arr: Array<string>) {
  if (0 < arr.length) {
    const elem = arr[0];
    assertType<undefined | string>(elem);
    console.log(elem);
  }
}
```

Therefore, it makes more sense to use a different pattern:

```
function logElemAt0(arr: Array<string>) {
  if (0 in arr) {
    const elem = arr[0];
    assertType<string>(elem);
    console.log(elem);
  }
}
```

8.7.8 Type checking options that have good defaults

By default, the following options produce warnings in editors, but we can also choose to produce compiler errors or ignore problems:

- `allowUnreachableCode`
- `allowUnusedLabels`
- `noUnusedLocals`
- `noUnusedParameters`

8.8 Compiling TypeScript with tools other than tsc

The TypeScript compiler tsc performs three tasks:

1. Type checking
2. Emitting JavaScript files
3. Emitting declaration files

External tools have become popular that do #2 and #3 much faster. The following subsections describe configuration options that help those tools.

8.8.1 Using tsc only for type checking

```
"compilerOptions": {
  "noEmit": true,
}
```

Sometimes, we want to use tsc only for type checking – e.g., if we run TypeScript directly or use external tools for compiling TypeScript files (to JavaScript files, declaration files, etc.):

- `noEmit`: If `true`, we can run tsc and it will only type-check the TypeScript code, it won't emit any files.

In principle, you don't have to provide output-related settings such as `rootDir` and `outDir` anymore. However, some external tools may need them.

8.8.2 Generating `.js` files via type stripping: `erasableSyntaxOnly` and `verbatimModuleSyntax`

```
"compilerOptions": {
  "erasableSyntaxOnly": true,
  "verbatimModuleSyntax": true, // implies "isolatedModules"
}
```

Type stripping is a simple and fast way of compiling TypeScript to JavaScript. It's what Node.js uses when it runs TypeScript. Type stripping is fast because it only supports a subset of TypeScript where two things are possible:

1. Type syntax can be detected and removed by only parsing the syntax – without performing additional semantic analyses.
2. No non-type language features are transpiled. In other words: Removing the type syntax is enough to produce JavaScript.

To help with type stripping, TypeScript has two compiler options that report errors if we use unsupported features:

- `verbatimModuleSyntax` forbids features that prevent #1.
- `erasableSyntaxOnly` forbids features that are transpiled.

Note that these options don't change what is emitted by tsc.

Useful related knowledge: "Type stripping technique: replacing types with spaces" (§6.5.1.1).

8.8.3 `erasableSyntaxOnly`: no transpiled language features

The compiler option `erasableSyntaxOnly` helps with type stripping. It forbids non-type TypeScript features that are not "current" JavaScript (as supported by the target platforms) and have to be transpiled. These are the most important ones:

- JSX
- Enums
- Parameter properties in class constructors.

- Namespaces
- Future JavaScript that is compiled to current JavaScript

Another feature that is forbidden by `erasableSyntaxOnly` is the legacy way of casting via angle brackets – because its syntax makes type stripping impossible in some cases (source):

```
<someType>someValue // not allowed
```

However, the alternative `as` is always better anyway:

```
someValue as someType // allowed
```

8.8.4 `verbatimModuleSyntax`: enforcing `type` in imports and exports

The compiler option `verbatimModuleSyntax` forces us to add the keyword `type` to type-only imports and exports.

When compiling TypeScript to JavaScript via type stripping, we need to remove the TypeScript parts. Most of those parts are easy to detect. The exception are imports and exports – e.g., without semantic analysis, we don't know if an import is a (TypeScript) type or a (JavaScript) value. If type-only imports and exports are marked with the keyword `type`, no such analysis is necessary.

Importing types

This is what the keyword `type` looks like in imports:

```
// Input: TypeScript
import { type SomeInterface, SomeClass } from './my-module.js';

// Output: JavaScript
import { SomeClass } from './my-module.js';
```

Note that a class is both a value and a type. In that case, no `type` keyword is needed because that part of the syntax can stay in plain JavaScript.

We can also apply `type` to the whole import:

```
import type { Type1, Type2, Type3 } from './types.js';
```

Exporting types

Inline type exports:

```
export type MyType = {};
export interface MyInterface {}
```

Export clauses:

```
type Type1 = {};
type Type2 = {};
export {
  type Type1,
  type Type2,
```

```
}

type Type3 = {};
type Type4 = {};
export type {
  Type3,
  Type4,
}
```

Alas, default-exporting only works for interfaces:

```
export default interface DefaultInterface {} // OK

type DefaultType = {}
export default DefaultType; // error
export default type DefaultType; // error

export default type {} // error
```

We can use the following workaround:

```
type DefaultType = {}
export {
  type DefaultType as default,
}
```

Why does this inconsistency exist? `type` is allowed as a (JavaScript-level) identifier after `export default`.

`isolatedModules`

Activating `verbatimModuleSyntax` also activates `isolatedModules`, which is why we only need the former setting. The latter prevents us from using some relatively obscure features that are also problematic.

As an aside, this option enables esbuild to compile TypeScript to JavaScript (source).

8.8.5 `isolatedDeclarations`: generating `.d.ts` files more efficiently

```
"compilerOptions": {
  // Only allowed if `declaration` or `composite` are true
  "isolatedDeclarations": true,
}
```

Option `isolatedDeclarations` helps external tools compile TypeScript files to declaration files, by forcing us to add more type annotations so that no type inference is needed for compilation (trivially simple type inference is still allowed – more on that soon). That has several benefits:

- The tools don't need to know the logic of type inference – which makes them simpler. Extracting declarations becomes a syntactic operation and doesn't really have to consider the type level.

- Not having to run type inference saves time.
- We can look at single files in isolation: Type inference sometimes has to visit other files to compute its results, which introduces dependencies on those files.
  - Incidentally, that explains the name of the compiler option.

`isolatedDeclarations` only produces compiler errors, it does not change what is emitted by tsc. It only affects constructs that are exported – because only those show up in declaration files. Module-internal code is not affected.

Let's look at three constructs affected by `isolatedDeclarations`.

Return types of top-level functions

For top-level functions, we should usually explicitly specify return types:

```
// OK: return type stated explicitly
export function f1(): string {
  return 123..toString();
}
// Error: return type requires inference
export function f2() {
  return 123..toString();
}
// OK: return type trivial to determine
export function f3() {
  return 123;
}
```

Types of variable declarations

More complicated variable declarations must have type annotations. Note that this only affects top-level declarations – e.g.: Variable declarations inside functions don't show up in declaration files and therefore don't matter.

```
// OK: type trivial to determine
export const value1 = 123;
// Error: type requires inference
export const value2 = 123..toString();
// OK: type stated explicitly
export const value3: string = 123..toString();
```

Types of class instance fields

Class instance fields must have type annotations (even though `tsc` can infer their types if there is an assignment in the constructor):

```
export class C {
  str: string; // required
  constructor(str: string) {
    this.str = str;
  }
}
```

`isolatedDeclarations` requires `declaration` or `composite`

I'd love to always use `isolatedDeclarations`, but TypeScript only allows it if option `declaration` or option `composite` are active. Jake Bailey explains why that is:

> At the implementation level, `isolatedDeclarations` diagnostics are extra declaration diagnostics produced by the declaration transformer, which we only run when `declaration` is enabled.
>
> Theoretically it could be implemented such that `isolatedDeclarations` enables those checks (the diagnostics actually come from us running the transformer and then throwing away the resulting AST), but it is a change from the original design.

Further reading

The TypeScript 5.5 release notes have a comprehensive section on isolated declarations.

8.9 Importing CommonJS from ESM

One key issue affects importing a CommonJS module from an ESM module:

- In ESM, the default export is the property `.default` of the module namespace object.
- In CommonJS, the module object *is* the default export – e.g., there are many CommonJS modules that set `module.exports` to a function.

Let's look at two options that help.

8.9.1 `allowSyntheticDefaultImports`: type-checking default imports of CommonJS modules

This option only affects type checking, not the JavaScript code emitted by TypeScript: If active, a default import of a CommonJS module refers to `module.exports` (not `module.exports.default`) – but only if there is no `module.exports.default`.

This reflects how Node.js handles default imports of CommonJS modules (source): "When importing CommonJS modules, the `module.exports` object is provided as the default export. Named exports may be available, provided by static analysis as a convenience for better ecosystem compatibility."

Do we need this option? Yes, but it's automatically activated if `moduleResolution` is `"bundler"` or if `module` is `"NodeNext"` (which activates `esModuleInterop` which activates `allowSyntheticDefaultImports`).

8.9.2 `esModuleInterop`: better compilation of TypeScript to CommonJS code

This option affects emitted CommonJS code:

- If `false`:
  - `import * as m from 'm'` is compiled to `const m = require('m')`.

  - `import m from 'm'` is (roughly) compiled to `const m = require('m')` and every access of `m` is compiled to `m.default`.
- If `true`:
  - `import * as m from 'm'` assigns a new object to `m` that has the same properties as `module.exports` plus a property `.default` that refers to `module.exports`.
  - `import m from 'm'` assigns a new object to `m` that has a single property `.default` that refers to `module.exports`. Every access of `m` is compiled to `m.default`.
- If a CommonJS module has the marker property `.__esModule` then it is always imported as if `esModuleInterop` were switched off.

Do we need this option? No, since we only author ESM modules.

8.10 One more option with a good default

We can usually ignore this option:

- `moduleDetection`: This option configures how TypeScript determines whether a file is a script or a module. It can usually be omitted because its default `"auto"` works well in most cases. You only need to explicitly set it to `"force"` if your codebase has a module that has neither imports nor exports. If `module` is `"NodeNext"` and `package.json` has `"type":"module"` then even those files are interpreted as modules.

8.11 Visual Studio Code

If you are unhappy with the module specifiers for local imports in automatically created imports then you can take a look at the following two settings:

```
javascript.preferences.importModuleSpecifierEnding
typescript.preferences.importModuleSpecifierEnding
```

By default, VSC should now be smart enough to add filename extensions where necessary.

8.12 Summary: Assemble your `tsconfig.json` by answering four questions

This is a starter `tsconfig.json` file with all settings. The following subsections explain which parts to remove – depending on your needs.

Alternatively, you can use my interactive tsconfig configurator via the command line or online.

```
{
  "include": ["src/**/*"],
  "compilerOptions": {
    // Specified explicitly (not derived from source file paths)
    "rootDir": "src",
    "outDir": "dist",

    //========== Target and module ==========
```

```
    // Nothing is ever transpiled
    "target": "ESNext", // sets up "lib" accordingly
    "module": "NodeNext", // sets up "moduleResolution"
    // Don't check .d.ts files
    "skipLibCheck": true,
    // Emptily imported modules must exist
    "noUncheckedSideEffectImports": true,
    // Allow importing JSON
    "resolveJsonModule": true,

    //========== Type checking ==========
    // Essential: activates several useful options
    "strict": true,
    // Beyond "strict": less important
    "exactOptionalPropertyTypes": true,
    "noFallthroughCasesInSwitch": true,
    "noImplicitOverride": true,
    "noImplicitReturns": true,
    "noPropertyAccessFromIndexSignature": true,
    "noUncheckedIndexedAccess": true,

    //========== Only JS at non-type level (type stripping etc.) ==========
    // Forbid non-JavaScript language constructs such as:
    // JSX, enums, constructor parameter properties, namespaces
    "erasableSyntaxOnly": true,
    // Enforce keyword `type` for type imports etc.
    "verbatimModuleSyntax": true, // implies "isolatedModules"

    //========== Use filename extension .ts in imports ==========
    "allowImportingTsExtensions": true,
    // Only needed if compiling to JavaScript
    "rewriteRelativeImportExtensions": true, // from .ts to .js

    //========== Emitted files ==========
    // tsc only type-checks, doesn't emit any files
    "noEmit": true,
    //----- Output: .js -----
    "sourceMap": true, // .js.map files
    //----- Output: .d.ts -----
    "declaration": true, // .d.ts files
    // "Go to definition" jumps to TS source etc.
    "declarationMap": true, // .d.ts.map files
    // - Enforces constraints that enable efficient .d.ts generation:
    //   no inferred return types for exported functions etc.
    // - Even though this option would be generally useful, it requires
    //   that `declaration` and/or `composite` are true.
    "isolatedDeclarations": true,
  }
```

```
}
```

8.12.1 Do you want to transpile new JavaScript to older JavaScript?

TypeScript can transpile new JavaScript features to code that only uses older "target" features. That can help support older JavaScript engines. If that's what you want, you must change `"target"`:

```
"compilerOptions": {
  // Transpile new JavaScript to old JavaScript
  "target": "ES20YY", // sets up "lib" accordingly
}
```

8.12.2 Should TypeScript only allow JavaScript features at the non-type level?

In other words: Should all non-JavaScript syntax be erasable? If yes, then these are the main features that are forbidden: JSX, enums, constructor parameter properties, and namespaces.

The starter tsconfig only allows erasable syntax. If you want to use any of the aforementioned features, then remove section "Only JS at non-type level".

8.12.3 Which filename extension do you want to use in local imports?

The starter tsconfig enables `.ts` in imports. If you want to use `.js`, you can remove section "Use filename extension .ts in imports".

8.12.4 What files should tsc emit?

- If `tsc` should emit *some* files, remove property `"noEmit"`.
- If `tsc` should *not* emit JavaScript, remove subsection "Output: .js".
- If `tsc` should *not* emit declarations, remove subsection "Output: .d.ts".

If no files are emitted, you can remove the following properties:

- `"rootDir"`
- `"outDir"`

Emitted files:

File	`tsconfig.json`
`*.js`	Default (deactivated via `"noEmit": true`)
`*.js.map`	`"sourceMap": true`
`*.d.ts`	`"declaration": true`
`*.d.ts.map`	`"declarationMap": true`

Source maps (`.map`) are only emitted if their source files are emitted.

8.13 Further reading

8.13.1 `tsconfig.json` recommendations by other people

- Matt Pocock's "The TSConfig Cheat Sheet"
- Pelle Wessman's `base.json`
- Sindre Sorhus' `tsconfig.json`

8.13.2 Sources of this chapter

Some of the sources were already mentioned earlier. These are additional sources I used:

- The official TSConfig documentation
- Section "Path Rewriting for Relative Paths" in the TypeScript 5.7 announcement.
- The esbuild documentation makes interesting observations about compiling TypeScript.

Chapter 9

Publishing npm packages with TypeScript

9.1 File system layout . . . 98
  9.1.1 `.gitignore` . . . 98
  9.1.2 Unit tests . . . 99
9.2 `tsconfig.json` . . . 99
  9.2.1 Where does the output go? . . . 100
  9.2.2 Output . . . 100
9.3 `package.json` . . . 101
  9.3.1 Using `.js` for ESM modules . . . 101
  9.3.2 Which files should be uploaded to the npm registry? . . . 101
  9.3.3 Package exports . . . 102
  9.3.4 Package imports . . . 103
  9.3.5 Package scripts . . . 103
  9.3.6 Development dependencies . . . 104
  9.3.7 Bin scripts: shell commands written in JavaScript . . . 105
9.4 Linting npm packages . . . 105
9.5 Further reading . . . 106

In this chapter, we'll create an ESM-based library package for npm via TypeScript:

- The described setup has worked well for me since TypeScript 4.7 (2022-05-24).
- We'll only use tsc, but the setup is ready for other tools. For more information, see "Compiling TypeScript with tools other than tsc" (§8.8).
- If you want to create a package with an executable and not a library, check out "Bin scripts: shell commands written in JavaScript" (§9.3.7).

Example package: `@rauschma/helpers`

This package uses the setup described in this chapter.

9.1 File system layout

Our npm package has the following file system layout:

```
my-package/
  README.md
  LICENSE
  package.json
  tsconfig.json
  docs/
    api/
  src/
    test/
  dist/
    test/
```

Comments:

- It's usually a good idea to include a `README.md` and a `LICENSE`
- `package.json` describes the package and is described later.
- `tsconfig.json` configures TypeScript and is described later.
- `docs/api/` is for API documentation generated via TypeDoc. See "Documenting TypeScript APIs via doc comments and TypeDoc" (§11).
- `src/` is for the TypeScript source code.
- `src/test/` is for integration tests – tests that span multiple modules. More on unit tests soon.
  - Why don't we put `src/` and `test/` next to each other? That would have the negative consequence that output files would be more deeply nested in the project directory than input files (more information).
- `dist/` is where TypeScript writes its output.

9.1.1 `.gitignore`

I'm using Git for version control. This is my `.gitignore` (located inside `my-package/`)

```
node_modules
dist
.DS_Store
```

Why these entries?

- `node_modules`: The most common practice currently seems to be *not* to check in the `node_modules` directory.

- `dist`: The compilation output of TypeScript is not checked into Git, but it is uploaded to the npm registry. More on that later.
- `.DS_Store`: This entry is about me being lazy as a macOS user. Since it's only need on that operating system, you can argue that Mac people should add it via a global configuration setting and keep it out of project-specific gitignores.

9.1.2 Unit tests

I have started to put the unit tests for a particular module next to that module:

```
src/
  util.ts
  util_test.ts
```

Given that unit tests help with understanding how a module works, it's useful if they are easy to find.

Tip for tests: self-reference the package

If an npm package has `"exports"`, it can *self-reference* them via its package name:

```
// src/misc/errors.ts
import {helperFunc} from 'my-package/misc/errors.js';
```

The Node.js documentation has more information on self-referencing and notes: "Self-referencing is available only if `package.json` has `"exports"`, and will allow importing only what that `"exports"` (in the `package.json`) allows."

Benefits of self-referencing:

- It is useful for tests (which can demonstrate how importing packages would use the code).
- It checks if your package exports are set up properly.

9.2 `tsconfig.json`

"Summary: Assemble your `tsconfig.json` by answering four questions" (§8.12) helps us with creating a `tsconfig.json` file by asking us four questions. Let's answer these questions for our npm package:

- Q: Do you want to transpile new JavaScript to older JavaScript?
  - A: No. If we can constrain ourselves to JavaScript features supported by our target platforms then the output is simpler and more similar to the input.
- Q: Should TypeScript only allow JavaScript features at the non-type level?
  - A: Yes because that keeps things simple and lets us use type stripping should we want to. It's a forward-looking way of writing TypeScript.
- Q: Which filename extension do you want to use in local imports?
  - A: ESM requires filename extensions and TypeScript traditionally does not change module specifiers during transpilation. Therefore, TypeScript ESM

code initially used `.js`. Now TypeScript transpilation can change `.ts` to `.js`. Therefore, we can use `.ts` – with the benefit that the same code also runs without transpilation on some platforms (Node.js, Deno, Bun, etc.). The only caveat is that we still have to use `.js` when we self-reference modules (see previous section) or use `import()`.

- Q: What files should tsc emit?
  - A: `.js`, `.js.map`, `.d.ts`, `.d.ts.map` (more information on that soon)

9.2.1 Where does the output go?

`tsconfig.json`:

```
{
  "include": ["src/**/*"],
  "compilerOptions": {
    "rootDir": "src",
    "outDir": "dist",
    // ···
  }
}
```

Consequences of these settings:

- Input: `src/util.ts`
  - Output: `dist/util.js`
- Input: `src/test/integration_test.ts`
  - Output: `dist/test/integration_test.js`

What if we want `src/` and `test/` to sit next to each other? See "Putting `src/` and `test/` next to each other" (§8.4.1.1) for more information.

9.2.2 Output

Given a TypeScript file `util.ts`, tsc writes the following output to `dist/`:

```
src/
  util.ts
dist/
  util.js
  util.js.map
  util.d.ts
  util.d.ts.map
```

Purposes of these files:

- `util.js`: JavaScript code contained in `util.ts`
- `util.js.map`: *source map* for the JavaScript code. It enables the following functionality when running `util.js`:
  - In a debugger, we see the TypeScript code.
  - Stack traces contain TypeScript source code locations.

- `util.d.ts`: types defined in `util.ts`
- `util.d.ts.map`: *declaration map* – a source map for `util.d.ts`. It enables TypeScript editors that support it to (e.g.) jump to the TypeScript source code of the definition of a type. I find that useful for libraries. It's why I include the TypeScript source in their packages.

9.3 package.json

Some settings in `package.json` also affect TypeScript. We'll look at those next. Related material:

- Chapter "Packages: JavaScript's units for software distribution" of "Shell scripting with Node.js" provides a comprehensive look at npm packages.
- You can also take a look the the `package.json` of `@rauschma/helpers`.

9.3.1 Using .js for ESM modules

By default, `.js` files are interpreted as CommonJS modules. The following setting lets us use that filename extension for ESM modules:

```
"type": "module",
```

9.3.2 Which files should be uploaded to the npm registry?

We have to specify which files should be uploaded to the npm registry. While there is also the `.npmignore` file, explicitly listing what's *included* is safer. That is done via the `package.json` property `"files"`:

```
"files": [
  "package.json",
  "README.md",
  "LICENSE",

  "src/**/*.ts",

  "dist/**/*.js",
  "dist/**/*.js.map",
  "dist/**/*.d.ts",
  "dist/**/*.d.ts.map",

  "!src/test/",
  "!src/**/*_test.ts",

  "!dist/test/",
  "!dist/**/*_test.js",
  "!dist/**/*_test.js.map",
  "!dist/**/*_test.d.ts",
```

```
    "!dist/**/*_test.d.ts.map"
  ],
```

In `.gitignore`, we have ignored directory `dist/` because it contains information that can be generated automatically. However, here it is explicitly included because most of its contents have to be in the npm package.

Patterns that start with exclamation marks (`!`) define which files to exclude. In this case, we exclude the tests:

- Some of them sit next to modules in `src/`.
- The remaining tests are located in `src/test/`.

9.3.3 Package exports

If we want a package to support old code, there are several `package.json` properties, we have to take into consideration:

- `"main"`: previously used by Node.js
- `"module"`: previously used by bundlers
- `"types"`: previously used by TypeScript
- `"typesVersions"`: previously used by TypeScript

In contrast, for modern code, we only need:

```
  "exports": {
    // Package exports go here
  },
```

Before we get into details, there are two questions we have to consider:

- Is our package only going to be imported via a bare import or is it going to support subpath imports?

  ```
  import {someFunc} from 'my-package'; // bare import
  import {someFunc} from 'my-package/sub/path'; // subpath import
  ```

- If we export subpaths: Are they going to have filename extensions or not?

Tips for answering the latter question:

- The extensionless style has a long tradition. That hasn't changed much with ESM, even though it requires filename extensions for local imports.

- Downside of the extensionless style (quoting the Node.js documentation): "With import maps now providing a standard for package resolution in browsers and other JavaScript runtimes, using the extensionless style can result in bloated import map definitions. Explicit file extensions can avoid this issue by enabling the import map to utilize a packages folder mapping to map multiple subpaths where possible instead of a separate map entry per package subpath export. This also mirrors the requirement of using the full specifier path in relative and absolute import specifiers."

This is how I currently decide:

- Most of my packages don't have any subpaths at all.

- If the package is a collection of modules, I export them with extensions.
- If the modules are more like different versions of the package (think synchronous vs. asynchronous) then I export them without extensions.

However, I don't have strong preferences and may change my mind in the future.

Specifying package exports

```
// Bare export
".": "./dist/main.js",

// Subpaths with extensions
"./misc/errors.js": "./dist/misc/errors.js", // single file
"./misc/*": "./dist/misc/*", // subtree

// Extensionless subpaths
"./misc/errors": "./dist/misc/errors.js", // single file
"./misc/*": "./dist/misc/*.js", // subtree
```

Notes:

- If there aren't many modules then multiple single-file entries are more self-explanatory than one subtree entry.
- By default, `.d.ts` files must sit next to `.js` files. But that can be changed via the `types` import condition.

For more information on this topic, see section "Package exports: controlling what other packages see" in "Exploring JavaScript".

9.3.4 Package imports

Node's package imports are also supported by TypeScript. They let us define aliases for paths. Those aliases have the benefit that they start at the top level of the package. This is an example:

```
"imports": {
  "#root/*": "./*"
},
```

We can use this package import as follows:

```
import pkg from '#root/package.json' with { type: 'json' };
console.log(pkg.version);
```

Package imports are especially helpful when the JavaScript output files are more deeply nested than the TypeScript input files. In that case we can't use relative paths to access files at the top level.

For more information on package imports, see the Node.js documentation.

9.3.5 Package scripts

Package scripts lets us define aliases such as `build` for shell commands and execute them

via `npm run build`. We can get a list of those aliases via `npm run` (without a script name).

These are commands I find useful for my library projects:

```
"scripts": {
  "\n========== Building ==========": "",
  "build": "npm run clean && tsc",
  "watch": "tsc --watch",
  "clean": "shx rm -rf ./dist/*",
  "\n========== Testing ==========": "",
  "test": "mocha --enable-source-maps --ui qunit",
  "testall": "mocha --enable-source-maps --ui qunit \"./dist/**/*_test.js\"",
  "\n========== Publishing ==========": "",
  "publishd": "npm publish --dry-run",
  "prepublishOnly": "npm run build"
},
```

Explanations:

- `build`: I clear directory `dist/` before each build. Why? When renaming TypeScript files, the old output files are not deleted. That is especially problematic with test files and regularly bites me. Whenever that happens, I can fix things via `npm run build`.
- `test`, `testall`:
  - `--enable-source-maps` enables source map support in Node.js and therefore accurate line numbers in stack traces.
  - The test runner Mocha supports several testing styles. I prefer `--ui qunit` (example).
- `publishd`: We publish an npm package via `npm publish`. `npm run publishd` invokes the "dry run" version of that command that doesn't make any changes but provides helpful feedback – e.g., it shows which files are going to be part of the package.
- `prepublishOnly`: Before `npm publish` uploads files to the npm registry, it invokes this script. By building before publishing, we ensure that no stale files and no old files are uploaded.

Why the named separators? The make the output of `npm run` easier to read.

9.3.6 Development dependencies

Even if a package of mine has no normal dependencies, it tends to have the following development dependencies:

```
"devDependencies": {
  "@types/mocha": "^10.0.6",
  "@types/node": "^20.12.12",
  "mocha": "^10.4.0",
  "shx": "^0.3.4",
  "typedoc": "^0.27.6"
},
```

Explanations:

- `@types/node`: In unit tests, I'm using `node:assert` for assertions such as `assert.deepEqual()`. This dependency provides types for that and other Node modules.
- `shx`: provides cross-platform versions of Unix shell commands. I'm often using:

  ```
  shx rm -rf
  shx chmod u+x
  ```

I also install the following two command line tools locally inside my projects so that they are guaranteed to be there. The neat thing about `npm run` is that it adds locally installed commands to the shell path – which means that they can be used in package scripts as if they were installed globally.

- `mocha` and `@types/mocha`: I still prefer Mocha's API and CLI user experience but Node's built-in test runner has become an interesting alternative.
- `typedoc`: I'm using TypeDoc to generate API documentation.

9.3.7 Bin scripts: shell commands written in JavaScript

A `package.json` can contain the property `"bin"` which sets up executables. Check out my project TSConfigurator for a full example. That project has the following property in `package.json`:

```
"bin": {
  "tsconfigurator": "./dist/tsconfigurator.js"
},
```

We can constrain the Node.js versions with which the bin scripts can be used:

```
"engines": {
  "node": ">=23.6.0"
},
```

The following package script is useful (invoked from `"build"`, after `"tsc"`):

```
"scripts": {
  ...
  "chmod": "shx chmod u+x ./dist/tsconfigurator.js"
}
```

If a package provides an executable and not a library, we don't need to emit `.d.ts` files. If we use type stripping for `.js`, we may not need `.js.map` files either.

9.4 Linting npm packages

Linting the public interfaces of packages:

- publint: "Lints npm packages to ensure the widest compatibility across environments, such as Vite, Webpack, Rollup, Node.js, etc."
- arethetypeswrong: "This project attempts to analyze npm package contents for issues with their TypeScript types, particularly ESM-related module resolution issues."
- npm-package-json-lint: "Configurable linter for `package.json` files"

- vitest-package-exports: "[…] get all exported APIs of a package and prevent unintended breaking changes". Despite its name, this tool does not require Vitest.

Linting npm packages internally:

- installed-check: "Verifies that installed modules comply with the requirements [the Node.js `"engines"` version range] specified in `package.json`."
- Knip: "Finds and fixes unused files, dependencies and exports." Supports JavaScript and TypeScript.
- Node Modules Inspector: "Visualize your `node_modules`, inspect dependencies, and more."
- Madge: create a visual graph of module dependencies, find circular dependencies, and more.

9.5 Further reading

- JavaScript modules (ESM): Chapter "Modules" in "Exploring JavaScript"
- npm packages: Chapter "Packages: JavaScript's units for software distribution" in "Shell scripting with Node.js"

Also useful:

- Chapter "Modules: Packages" of the Node.js documentation
- Section "`package.json "exports"`" of the TypeScript Handbook

Chapter 10

Creating apps with TypeScript

10.1 Writing TypeScript apps for web browsers 107
10.2 Writing TypeScript apps for server-side runtimes 107

In this chapter, I'll give a few tips for using TypeScript to write apps.

10.1 Writing TypeScript apps for web browsers

Even with plain JavaScript, most projects use build tools for frontend web apps, for various reasons. Most of those build tools support TypeScript out of the box.

If you:

- Want to write a frontend app with TypeScript
- Have a particular frontend framework (or vanilla code) in mind
- Don't want to think too much about tooling

Then you can take a look at the "Getting Started" page for Vite, a popular frontend build tool.

But there are many other build tools for JavaScript. These are some of them:

- Smaller scope: esbuild, Rolldown, Rollup, Rspack, tsup, Turbopack
- More comprehensive: Parcel, Rsbuild, Vite
- Monorepo tools: Nx, Turborepo

10.2 Writing TypeScript apps for server-side runtimes

- Node.js, Deno and Bun can all run TypeScript directly.

- By default, Node.js only supports type stripping, but experimental support for non-erasable features such as JSX and enums is available via `--experimental-transform-types`. Type stripping will remain the default.

Getting started with Node.js:

- My GitHub repository `nodejs-type-stripping` demonstrates how type stripping works in Node.js.
- "Publishing npm packages with TypeScript" (§9) explains how to transpile TypeScript for Node.js.

Chapter 11

Documenting TypeScript APIs via doc comments and TypeDoc

11.1 Doc comments . . . 109
11.1.1 JSDoc comments . . . 109
11.1.2 TSDoc comments . . . 110
11.1.3 TypeDoc . . . 110
11.2 Generating documentation . . . 110
11.3 Referring to parts of files from doc comments . . . 111
11.3.1 Why is that useful? . . . 111
11.3.2 Real-world example . . . 111
11.4 Further reading . . . 112

This chapter describes how to document TypeScript APIs via *doc comments* (multi-line comments whose contents follow a standard format). We will use the npm-installable command line tool TypeDoc for this task.

11.1 Doc comments

11.1.1 JSDoc comments

The de-facto standard for documenting APIs in JavaScript are JSDoc comments (which were inspired by Javadoc comments). They are multi-line comments where the opening delimiter is followed by a second asterisk:

```
/**
 * Adds two numbers
 *
 * @param {number} x - The first operand
 * @param {number} y - The second operand
```

```
 * @returns {number} The sum of both operands
 */
function add(x: number, y: number): number {
  return x + y;
}
```

The hyphen after parameter names such as x is optional. TypeScript itself supports JSDoc comments as a way to specify types in plain JavaScript code (more information).

11.1.2 TSDoc comments

TSDoc adapts JSDoc comments so that they are a better fit for TypeScript – e.g., types in comments are not allowed and the hyphen is mandatory:

```
/**
 * Adds two numbers
 *
 * @param x - The first operand
 * @param y - The second operand
 * @returns The sum of both operands
 */
function add(x: number, y: number): number {
  return x + y;
}
```

11.1.3 TypeDoc

The API documentation generator TypeDoc uses doc comments to generate HTML API documentation. TSDoc comments are preferred, but JSDoc comments are supported, too. TypeDoc's features include:

- Support for Markdown in doc comments, including syntax highlighting for code blocks.
- Doc comments can refer to code fragments in external files – which can be tested more easily.
- Web pages generated from files written in Markdown ("external documents").

11.2 Generating documentation

We can use the following `package.json` script to convert TSDoc comments to API documentation:

```
"scripts": {
  "\n========== TypeDoc ==========": "",
  "api": "shx rm -rf docs/api/ && typedoc --out docs/api/ --readme none
  --entryPoints src --entryPointStrategy expand --exclude '**/*_test.ts'",
},
```

The entry for `"api"` is a single line; I have broken it up so that it can be displayed better.

As a complementary measure, we can serve GitHub pages from `docs/`:

- File in repository: `my-package/docs/api/index.html`
- File online (user `robin`): `https://robin.github.io/my-package/api/index.html`

You can check out the API docs for `@rauschma/helpers` online (warning: still underdocumented).

11.3 Referring to parts of files from doc comments

Since version 0.27.7, TypeDoc lets us refer to parts of external files via the doc tag `{@includeCode}`:

File `util.ts`:

```
/**
 * {@includeCode ./util_test.ts#utilFunc}
 */
function utilFunc(): void {}
```

Note the hash (`#`) and the name `utilFunc` after the path of the file: It refers to a *region* inside `util_test.ts`. A region is a way to mark sequences of lines in a source file via comments. Regions are also supported by Visual Studio Code where they can be folded (documentation).

This is what the region inside `util_test.ts` looks like:

```
test('utilFunc', () => {
  //#region utilFunc
  // ...
  //#endregion utilFunc
});
```

11.3.1 Why is that useful?

The file names already suggest the use case for this feature: It enables us to publish documentation where all code examples (such as region `utilFunc`) are tested.

In the past, TypeDoc only let us include full files, which meant one file per example – with test boilerplate showing up in the documentation.

11.3.2 Real-world example

- The following files are an excerpt from real-world code.
- Check out the result online: Function arrayToChunks
  - Another example: Class UnexpectedValueError

File `array.ts`:

```
/**
 * Split `arr` into chunks with length `chunkLen` and return them
 * in an Array.
 * {@includeCode ./array_test.ts#arrayToChunks}
 */
```

```
export function arrayToChunks<T>(
  arr: Array<T>, chunkLen: number
): Array<Array<T>> {
  // ···
}
```

File `array_test.ts`:

```
// ···
test('arrayToChunks', () => {
  //#region arrayToChunks
  const arr = ['a', 'b', 'c', 'd'];
  assert.deepEqual(
    arrayToChunks(arr, 1),
    [['a'], ['b'], ['c'], ['d']],
  );
  //#endregion arrayToChunks

  assert.deepEqual(
    arrayToChunks(arr, 2),
    [['a', 'b'], ['c', 'd']],
  );
});
```

11.4 Further reading

- Documentation for `{@includeCode}`
- Related tools:
  - Markcheck checks code example in Markdown files.
  - jsdoctest runs JSDoc examples (written in JavaScript) as doctests.

Chapter 12

Strategies for migrating to TypeScript

12.1 Strategy: mixed JavaScript/TypeScript code bases . . . 113
12.2 Strategy: adding type information to plain JavaScript files . . . 114
12.3 Strategy: linting before activating a compiler option . . . 114
12.4 Strategy: Too many errors? Use snapshot testing . . . 115
12.5 Tools that help with migrating to TypeScript . . . 115
12.6 Conclusion and further reading . . . 115

This chapter gives an overview of strategies for migrating code bases from JavaScript to TypeScript. It also mentions material for further reading.

12.1 Strategy: mixed JavaScript/TypeScript code bases

The TypeScript compiler supports a mix of JavaScript and TypeScript files if we use the compiler option `--allowJs`:

- TypeScript files are compiled.
- JavaScript files are simply copied over to the output directory (after a few simple type checks).

At first, there are only JavaScript files. Then, one by one, we switch files to TypeScript. While we do so, our code base keeps being compiled.

This is what `tsconfig.json` looks like:

```
{
  "compilerOptions": {
    ...
    "allowJs": true
```

```
  }
}
```

More information:

- "Incrementally Migrating JavaScript to TypeScript" by Clay Allsopp.

12.2 Strategy: adding type information to plain JavaScript files

This approach works as follows:

- We continue to use our current build infrastructure.
- We run the TypeScript compiler, but only as a type checker (compiler option `--noEmit`). In addition to the compiler option `--allowJs` (for allowing and copying JavaScript files), we also have to use the compiler option `--checkJs` (for type-checking JavaScript files).
- We add type information via JSDoc comments (see example below) and declaration files.
- Once TypeScript's type checker doesn't complain anymore, we use the compiler to build the code base. Switching from `.js` files to `.ts` files is not urgent now because the whole code base is already fully statically typed. We can even produce type files (filename extension `.d.ts`) now.

This is how we specify static types for plain JavaScript via JSDoc comments:

```
/**
 * @param {number} x - The first operand
 * @param {number} y - The second operand
 * @returns {number} The sum of both operands
 */
function add(x, y) {
  return x + y;
}

/** @typedef {{ prop1: string, prop2: string, prop3?: number }} SpecialType */
/** @typedef {(data: string, index?: number) => boolean} Predicate */
```

More information:

- "Type-checking JavaScript files" (§6.8)
- "How we gradually migrated to TypeScript at Unsplash" by Oliver Joseph Ash

12.3 Strategy: linting before activating a compiler option

Projects that migrate to TypeScript often start with the default type checking and then add more checks, in stages – e.g.:

1. Initially: default checks
2. Activate `strictNullChecks` (prevents `undefined` and `null` from being used unless the type is `T | undefined` or `T | null`, respectively)

3. Activate `noImplicitAny` (prevents omitting types for parameters and more)
4. Activate `strict` (includes the previous two compiler options – which can be removed – and adds additional ones)

In contrast to compiler options, we can activate linting options on a per-file basis. This helps when switching from less strict type checking to stricter type checking: We can lint before making the switch. These are examples of useful rules that the TypeScript linter typescript-eslint provides:

- Preparing for `noImplicitAny`:
  - `no-unsafe-call`
  - `no-unsafe-member-access`
  - `no-unsafe-argument`
  - `no-unsafe-assignment`
  - `no-explicit-any`
- Preparing for `erasableSyntaxOnly`:
  - `eslint-plugin-erasable-syntax-only`

12.4 Strategy: Too many errors? Use snapshot testing

In large JavaScript projects, switching to TypeScript may produce too many errors – no matter which approach we choose. Then snapshot-testing the TypeScript errors may be an option:

- We run the TypeScript compiler on the whole code base for the first time.
- The errors produced by the compiler become our initial snapshot.
- As we work on the code base, we compare new error output with the previous snapshot:
  - Sometimes existing errors disappear. Then we can create a new snapshot.
  - Sometimes new errors appear. Then we either have to fix these errors or create a new snapshot.

More information:

- "How to Incrementally Migrate 100k Lines of Code to Typescript" by Dylan Vann

12.5 Tools that help with migrating to TypeScript

- ts-migrate helps with the first step of moving from JavaScript to TypeScript: "The resulting code will pass the build, but a followup is required to improve type safety. There will be lots of `//@ts-expect-error`, and `any` that will need to be fixed over time. In general, it is a lot nicer than starting from scratch."

- type-coverage shows which percentage of identifiers have the type `any`.

12.6 Conclusion and further reading

We have taken a quick look at strategies for migrating to TypeScript. Two more tips:

- Start your migration with experiments: Play with your code base and try out various strategies before committing to one of them.
- Then lay out a clear plan for going forward. Discuss prioritization with your team:
  - Sometimes finishing the migration quickly may take priority.
  - Sometimes the code remaining fully functional during the migration may be more important.
  - And so on…

Further reading:

- "Migrating from JavaScript" in the TypeScript Handbook

Part IV

Basic types

Chapter 13

What is a type in TypeScript? Two perspectives

13.1 Two questions for each perspective 119
13.2 Dynamic perspective: a type is a set of values 119
13.3 Static perspective: relationships between types 120
13.4 Nominal type systems vs. structural type systems 120
13.5 Further reading 121

What are types in TypeScript? This chapter describes two perspectives that help with understanding them. Both are useful; they complement each other.

13.1 Two questions for each perspective

The following two questions are important for understanding how types work and need to be answered from each of the two perspectives.

1. What does it mean for `arg` to have the type `MyType`?

```
function myFunc(arg: MyType): void {}
```

2. How is `UnionType` derived from `Type1` and `Type2`?

```
type UnionType = Type1 | Type2;
```

13.2 Dynamic perspective: a type is a set of values

From this perspective, we are interested in values and a type is a set of values:

1. We can pass a given value to `myFunc()` if it is included in `MyType`.
2. `UnionType` (a set) is defined as the set-theoretic union of `Type1` and `Type2`.

13.3 Static perspective: relationships between types

From this perspective:

- The source code has locations and each location has a static type. In a TypeScript-aware editor, we can see the static type of a location if we hover above it with the cursor.
- Types are defined via their relationships with other types.

The most important type relationship is *assignment compatibility*: Can a location whose type is `Src` be assigned to a location whose type is `Trg`? The answer is yes if:

- `Src` and `Trg` are identical types.
- `Src` or `Trg` is the type `any`.
- `Src` is a string literal type and `Trg` is the primitive type `string`.
- `Src` is a union type and each constituent type of `Src` is assignable to `Trg`.
- `Src` is an intersection type and at least one constituent type of `Src` is assignable to `Trg`.
- `Trg` is a union type and `Src` is assignable to at least one constituent type of `Trg`.
- `Trg` is an intersection type and `Src` is assignable to each constituent type of `Trg`.
- Etc.

Let's consider the questions:

1. Parameter `arg` having type `MyType` means that we can only pass a value to `myFunc()` whose type is assignable to `MyType`.
2. `UnionType` is defined by the relationships it has with other types. Above, we have seen two rules for union types.

13.4 Nominal type systems vs. structural type systems

One of the responsibilities of a static type system is to determine if two static types are compatible – e.g.:

- The static type `Src` of an actual parameter (e.g., provided via a function call)
- The static type `Trg` of the corresponding formal parameter (e.g., specified as part of a function definition)

The type system needs to check if `Src` is assignable to `Trg`. Two approaches for this check are (roughly):

- In a *nominal* or *nominative* type system, two static types are equal if they have the same identity ("name"). `Src` is only assignable to `Trg` if they are equal or if a relationship between them was specified explicitly – e.g., an inheritance relationship (`extends`).
  - Languages with nominal type systems include C++, Java and C#.
- In a *structural* type system, a type `Src` is assignable to a type `Trg` if `Trg` has a structure that can receive what's in `Src` — e.g.: For each field `Src.F`, there must be a field `Trg.F` such that `Src.F` is assignable to `Trg.F`.

  - Languages with structural type systems include TypeScript, Go (interfaces) and OCaml (objects).

The following code produces a type error in the last line with a nominal type system, but is legal with TypeScript's structural type system because class `A` and class `B` have the same structure:

```
class A {
  typeName = 'A';
}
class B {
  typeName = 'B';
}
const someVariable: A = new B();
```

TypeScript's interfaces also work structurally – they don't have to be implemented in order to match:

```
interface HasTypeName {
  typeName: string;
}
const hasTypeName: HasTypeName = new A(); // OK
```

13.5 Further reading

- Chapter "Type Compatibility" in the TypeScript Handbook
- TypeScript Language Specification 1.8: TypeScript originally had a formal language specification but it was discontinued after TypeScript 1.8 (which came out in 2016). It has since been removed from TypeScript's repositories, but a PDF file can still be downloaded from an old commit. Especially helpful: section "3.11 Type Relationships".

Chapter 14

The top types `any` and `unknown`

14.1 TypeScript's two top types . . . 123
14.2 The top type `any` . . . 123
  14.2.1 Example: `JSON.parse()` . . . 124
  14.2.2 Example: `String()` . . . 124
  14.2.3 The compiler option `noImplicitAny` . . . 125
14.3 The top type `unknown` . . . 125

In TypeScript, `any` and `unknown` are types that contain all values. In this chapter, we examine what they are and what they can be used for.

14.1 TypeScript's two top types

`any` and `unknown` are so-called *top types* in TypeScript. Quoting Wikipedia:

> The *top type* […] is the *universal* type, sometimes called the *universal supertype* as all other types in any given type system are subtypes […]. In most cases it is the type which contains every possible [value] in the type system of interest.

That is, when viewing types as sets of values (for more information on what types are, see "What is a type in TypeScript? Two perspectives" (§13)), `any` and `unknown` are sets that contain all values.

TypeScript also has the *bottom type* `never`, which is the empty set and explained in "The bottom type `never`" (§15).

14.2 The top type `any`

If a value has type `any`, we can do everything with it:

```
function func(value: any) {
  // Only allowed for numbers, but they are a subtype of `any`
  5 * value;

  // Normally the type signature of `value` must contain .propName
  value.propName;

  // Normally only allowed for Arrays and types with index signatures
  value[123];
}
```

Every type is assignable to type `any`:

```
let storageLocation: any;

storageLocation = null;
storageLocation = true;
storageLocation = {};
```

Type `any` is assignable to every type:

```
function func(value: any) {
  const a: null = value;
  const b: boolean = value;
  const c: object = value;
}
```

With `any` we lose any protection that is normally given to us by TypeScript's static type system. Therefore, it should only be used as a last resort, if we can't use more specific types or `unknown`.

14.2.1 Example: `JSON.parse()`

The result of `JSON.parse()` depends on dynamic input, which is why the return type is `any` (I have omitted the parameter `reviver` from the signature):

```
JSON.parse(text: string): any;
```

`JSON.parse()` was added to TypeScript before the type `unknown` existed. Otherwise, its return type would probably be `unknown`.

14.2.2 Example: `String()`

The function `String()`, which converts arbitrary values to strings, has the following type signature:

```
interface StringConstructor {
  (value?: any): string; // call signature
  // ···
}
```

14.2.3 The compiler option `noImplicitAny`

If the compiler option `noImplicitAny` is `true`, TypeScript requires explicit type annotations in locations where it can't infer types. The most important example is parameters definitions. If this option is `false`, it (implicitly) uses the type `any` in those locations.

This is an example of a compiler error that is caused by `noImplicitAny` being `true`:

```
// @ts-expect-error: Parameter 'name' implicitly has an 'any' type.
function hello(name) {
  return `Hello ${name}!`
}
assertType<
  (name: any) => string
>(hello);
```

TypeScript does not complain about us using the type `any` explicitly:

```
function hello(name: any) {
  return `Hello ${name}!`
}
```

14.3 The top type `unknown`

The type `unknown` is a type-safe version of the type `any`. Whenever you are thinking of using `any`, try using `unknown` first. `unknown` is similar to `any` in that we can assign any value to it:

```
let storageLocation: unknown;

storageLocation = null;
storageLocation = true;
storageLocation = {};
```

However, an `unknown` value is not assignable to anything:

```
function func(value: unknown) {
  // @ts-expect-error: Type 'unknown' is not assignable to type 'null'.
  const a: null = value;
  // @ts-expect-error: Type 'unknown' is not assignable to type 'boolean'.
  const b: boolean = value;
  // @ts-expect-error: Type 'unknown' is not assignable to type 'object'.
  const c: object = value;
}
```

Therefore, if we have a value of type `unknown`, we must narrow that type before we can do anything with the value – e.g., via:

- Type assertions:

  ```
  function func(value: unknown) {
    // @ts-expect-error: 'value' is of type 'unknown'.
  ```

```
    value.toFixed(2);

    // Type assertion:
    (value as number).toFixed(2); // OK
  }
```

- Equality:

```
  function func(value: unknown) {
    // @ts-expect-error: 'value' is of type 'unknown'.
    value * 5;

    if (value === 123) { // equality
      assertType<123>(value);
      value * 5; // OK
    }
  }
```

- Type guards:

```
  function func(value: unknown) {
    // @ts-expect-error: 'value' is of type 'unknown'.
    value.length;

    if (typeof value === 'string') { // type guard
      assertType<string>(value);
      value.length; // OK
    }
  }
```

- Assertion functions:

```
  function func(value: unknown) {
    // @ts-expect-error: 'value' is of type 'unknown'.
    value.test('abc');

    assertIsRegExp(value); // assertion function

    assertType<RegExp>(value);
    value.test('abc'); // OK
  }

  /** An assertion function */
  function assertIsRegExp(arg: unknown): asserts arg is RegExp {
    if (! (arg instanceof RegExp)) {
      throw new TypeError('Not a RegExp: ' + arg);
    }
  }
```

Chapter 15

The bottom type `never`

15.1 `never` is a bottom type 127
15.2 `never` is the empty set 128
15.3 Use case for `never`: filtering union types 128
15.4 Use case for `never`: exhaustiveness checks at compile time 129
    15.4.1 Exhaustiveness checks and `if` 131
15.5 Use case for `never`: forbidding properties 131
15.6 Functions that return `never` 131
15.7 Sources of this chapter 132

In this chapter, we look at the special TypeScript type `never` which, roughly, is the type of things that never happen. As we'll see, it has a surprising number of applications.

15.1 `never` is a bottom type

If we interpret types as sets of values then:

- Type `Sub` is a subtype of type `Sup` (`Sub <: Sup`)
- if `Sub` is a subset of `Sup` (`Sub ⊆ Sup`).

Two kinds of types are special:

- A top type `T` includes all values and all types are subtypes of `T`.
- A bottom type `B` is the empty set and a subtype of all types.

In TypeScript:

- `any` and `unknown` are top types and explained in "The top types `any` and `unknown`" (§14).
- `never` is a bottom type.

15.2 never is the empty set

When computing with types, type unions are sometimes used to represent sets of (type-level) values. Then the empty set is represented by `never`:

```
type _ = [
  Assert<Equal<
    keyof {a: 1, b: 2},
    'a' | 'b' // set of types
  >>,
  Assert<Equal<
    keyof {},
    never // empty set
  >>,
];
```

Similarly, if we use the type operator `&` to intersect two types that have no elements in common, we get the empty set:

```
type _ = Assert<Equal<
  boolean & symbol,
  never
>>;
```

If we use the type operator `|` to compute the union of a type `T` and `never` then the result is `T`:

```
type _ = Assert<Equal<
  'a' | 'b' | never,
  'a' | 'b'
>>;
```

15.3 Use case for never: filtering union types

We can use conditional types to filter union types:

```
type KeepStrings<T> = T extends string ? T : never;
type _ = [
  Assert<Equal<
    KeepStrings<'abc'>, // normal instantiation
    'abc'
  >>,
  Assert<Equal<
    KeepStrings<123>, // normal instantiation
    never
  >>,
  Assert<Equal<
    KeepStrings<'a' | 'b' | 0 | 1>, // distributed instantiation
    'a' | 'b'
  >>,
];
```

We use two phenomena to make this work:

- When we apply a conditional type to a union type, it is *distributed* – applied to each element of the union.
- In the resulting union of types, the `never` types returned in the false branch of `KeepStrings` disappear (see previous section).

More information: "Filtering union types by conditionally returning `never`"

15.4 Use case for `never`: exhaustiveness checks at compile time

With type inference, TypeScript keeps track of what values a variable still can have – e.g.:

```
function f(x: boolean): void {
  assertType<false | true>(x); // (A)
  if (x === true) {
    return;
  }
  assertType<false>(x); // (B)
  if (x === false) {
    return;
  }
  assertType<never>(x); // (C)
}
```

In line A, `x` can still have the value `false` and `true`. After we return if `x` has the value `true`, it can still have the value `false` (line B). After we return if `x` has the value `false`, there are no more values this variable can have, which is why it has the type `never` (line C).

This behavior is especially useful for enums and unions used like enums because it enables *exhaustiveness checks* (checking if we have exhaustively handled all cases):

```
enum Color { Red, Green }
```

The following pattern works well for JavaScript because it checks at runtime if `color` has an unexpected value:

```
function colorToString(color: Color): string {
  switch (color) {
    case Color.Red:
      return 'RED';
    case Color.Green:
      return 'GREEN';
    default:
      throw new UnexpectedValueError(color);
  }
}
```

How can we support this pattern at the type level so that we get a warning if we accidentally don't consider all member of the enum `Color`? (The return type `string` also keeps us

safe but with the technique we are about to see, we even get protection if there is no return time. Additionally, we are also protected from illegal values at runtime.)

Let's first examine how the inferred value of `color` changes as we add cases:

```
function exploreSwitch(color: Color) {
  switch (color) {
    default:
      assertType<Color.Red | Color.Green>(color);
  }
  switch (color) {
    case Color.Red:
      break;
    default:
      assertType<Color.Green>(color);
  }
  switch (color) {
    case Color.Red:
      break;
    case Color.Green:
      break;
    default:
      assertType<never>(color);
  }
}
```

Once again, the type records what values `color` still can have.

The following implementation of the class `UnexpectedValueError` requires that the type of its actual argument be `never`:

```
class UnexpectedValueError extends Error {
  constructor(
    // Type enables type checking
    value: never,
    // Only solution that can stringify undefined, null, symbols, and
    // objects without prototypes
    message = `Unexpected value: ${{}.toString.call(value)}`
  ) {
    super(message)
  }
}
```

Now we get a compile-time warning if we forget a case because we have not eliminated all values that `color` can have:

```
function colorToString(color: Color): string {
  switch (color) {
    case Color.Red:
      return 'RED';
    default:
```

```
      assertType<Color.Green>(color);
      // @ts-expect-error: Argument of type 'Color.Green' is not
      // assignable to parameter of type 'never'.
      throw new UnexpectedValueError(color);
  }
}
```

15.4.1 Exhaustiveness checks and `if`

The exhaustiveness check also works if we handle cases via `if`:

```
function colorToString(color: Color): string {
  assertType<Color.Red | Color.Green>(color);
  if (color === Color.Red) {
    return 'RED';
  }
  assertType<Color.Green>(color);
  if (color === Color.Green) {
    return 'GREEN';
  }
  assertType<never>(color);
  throw new UnexpectedValueError(color);
}
```

15.5 Use case for `never`: forbidding properties

Given that no other type is assignable to `never`, we can use it to forbid properties – e.g. those with string keys:

```
type EmptyObject = Record<string, never>;

// @ts-expect-error: Type 'number' is not assignable to type 'never'.
const obj1: EmptyObject = { prop: 123 };
const obj2: EmptyObject = {}; // OK
```

For more information, see "Forbidding properties via `never`" (§18.6).

15.6 Functions that return `never`

`never` also serves as a marker for functions that never return – e.g.:

```
function throwError(message: string): never {
  throw new Error(message);
}
```

If we call such functions, TypeScript knows that execution ends and adjusts inferred types accordingly. For more information, see "Return type `never`: functions that don't return".

15.7 Sources of this chapter

- Section "Better Support for `never`-Returning Functions" in "Announcing TypeScript 3.7" by Daniel Rosenwasser for Microsoft
- Blog post "The `never` type and error handling in TypeScript" by Stefan Baumgartner

Chapter 16

Symbols in TypeScript

16.1 Types for symbols . . . 133
  16.1.1 `symbol` and `typeof MY_SYMBOL` . . . 133
  16.1.2 The type `unique symbol` . . . 134
16.2 Unions of symbol types . . . 136
16.3 Symbols as special values . . . 137
16.4 Symbols as enum values . . . 137
16.5 Further reading . . . 138

In this chapter, we examine how TypeScript handles JavaScript symbols at the type level.

If you want to refresh your knowledge of JavaScript symbols, you can check out chapter "Symbols" of "Exploring JavaScript".

16.1 Types for symbols

16.1.1 `symbol` and `typeof MY_SYMBOL`

Type inference usually gives us:

- broader, more general types for `let`
- narrower, more specific types for `const`

For example:

```
let value1 = 123;
assertType<number>(value1);

const value2 = 123;
assertType<123>(value2);
```

That is also true for symbols:

```
let SYM1 = Symbol('SYM1');
assertType<symbol>(SYM1);

const SYM2 = Symbol('SYM2');
assertType<typeof SYM2>(SYM2);
```

`symbol` is the type of all symbols. `typeof SYM2` is the type of one specific symbol. There is no way for us to create another value that matches `typeof SYM2`:

```
function f(_sym: typeof SYM2) {}

f(SYM2); // OK
// @ts-expect-error: Argument of type 'symbol' is not assignable to
// parameter of type 'unique symbol'.
f(Symbol('SYM2')); // new, different value!
```

Note the type `unique symbol` in the error message. We'll get to what it is soon.

Out inability to create a new symbol that is equal to the original `SYM2` is a JavaScript phenomenon:

```
> Symbol() === Symbol()
false
```

Symbols are similar to object literals in this regard:

```
> {} === {}
false
```

Pitfall: assignment broadens the type of a symbol

If we assign a variable `SYM` with the type `typeof SYM` to another variable `X`, then the type of the latter is broadened to `symbol` – even when we declare it with `const`.

```
const SYM = Symbol('SYM'); // typeof SYM

function getSym(): typeof SYM {
  const X = SYM; // symbol

  // @ts-expect-error: Type 'symbol' is not assignable to
  // type 'unique symbol'.
  return X;
}
```

Related GitHub issue: "`unique symbol` lost on assignment to `const` despite type assertion"

16.1.2 The type `unique symbol`

The type `unique symbol` is a subtype of `symbol`. It means that this particular location holds a symbol with a particular type. It is very similar to `typeof SOME_SYMBOL` but does not name the particular symbol. Each location of `unique symbol` is different and incompatible with all other locations (similar to `typeof SOME_SYMBOL`).

`unique symbol` can be used in `const` variable declaration and `static readonly` properties. If we want to express uniqueness elsewhere, we have to use `typeof S` – as we have done previously. Given that `unique symbol` is basically another way of expressing `typeof S`, it's not very useful

```
const SYM: unique symbol = Symbol('SYM');
class MyClass {
  static readonly SYM: unique symbol = Symbol('SYM');
}
```

The previous code is completely equivalent to:

```
const SYM = Symbol('SYM');
class MyClass {
  static readonly SYM = Symbol('SYM');
}
```

There is one more location where we can use `unique symbol` – as the type of a read-only property. That is done to declare the types of well-known symbols such as `Symbol.iterator` (file `lib.es2015.iterable.d.ts`):

```
interface SymbolConstructor {
  readonly iterator: unique symbol;
}
```

Why is the name of the interface `SymbolConstructor`? That's due to how symbols are set up in file `lib.es2015.symbol.d.ts`:

```
interface SymbolConstructor {
  readonly prototype: Symbol;
  (description?: string | number): symbol;
  for(key: string): symbol;
  keyFor(sym: symbol): string | undefined;
}

declare var Symbol: SymbolConstructor;
```

We can't create properties whose type is `unique symbol`

Consider the following type:

```
type Obj = {
  readonly sym: unique symbol,
};
```

We can't create an object that is assignable to `Obj`:

```
const obj1: Obj = {
  // @ts-expect-error: Type 'symbol' is not assignable to
  // type 'unique symbol'.
  sym: Symbol('SYM'),
};
```

```
const SYM: unique symbol = Symbol('SYM');
const obj2: Obj = {
  // @ts-expect-error: Type 'typeof SYM' is not assignable to
  // type 'typeof sym'.
  sym: SYM,
};
```

In the previous subsection `SymbolConstructor.iterator` was not meant for yet-to-be-created objects. It was meant for a single global value that already existed.

16.2 Unions of symbol types

We can use symbols to define union types:

```
const ACTIVE = Symbol('ACTIVE');
const INACTIVE = Symbol('INACTIVE');
type ActSym = typeof ACTIVE | typeof INACTIVE;

const activation1: ActSym = ACTIVE;
const activation2: ActSym = INACTIVE;
// @ts-expect-error: Type 'unique symbol' is not assignable to
// type 'ActSym'.
const activation3: ActSym = Symbol('ACTIVE');
```

We have to use `typeof` to go from program level to type level:

- Program level: `ACTIVE`
- Type level: `typeof ACTIVE`

How does a symbol-based union type compare to a string-based union type such as the one below?

```
type ActStr = 'ACTIVE' | 'INACTIVE';
```

To make it easier to compare `ActSym` with `ActStr`, let's define the latter in a more complicated way (which we normally wouldn't do):

```
const ACTIVE = 'ACTIVE';
const INACTIVE = 'INACTIVE';
type ActStr = typeof ACTIVE | typeof INACTIVE;

const activation1: ActStr = ACTIVE;
const activation2: ActStr = INACTIVE;
const activation3: ActStr = 'ACTIVE'; // OK!
```

What are the pros and cons of a string-based union type?

- Pro: No need to import constants, we can simply mention the strings (see last line above).
- Con: The union values are not type-safe. Strings are compared by value (not by identity), which is why a value can be mistaken to be a member of `ActStr` when it actually isn't. That kind of mistake cannot happen with symbol-based type unions.

16.3 Symbols as special values

`undefined` and `null` are often used as special "non-values" that are different from the actual values of a type:

```
type StreamValue =
  | null // end of file
  | string
;
```

However, a symbol can be a better, more self-explanatory alternative:

```
const EOF = Symbol('EOF');
type StreamValue =
  | typeof EOF
  | string
;
```

For more information on this topic, see "Adding special values to types" (§17).

16.4 Symbols as enum values

Symbols also work well if we want to create an enum with new, unique values:

```
const Active = Symbol('Active');
const Inactive = Symbol('Inactive');

const Activation = {
  __proto__: null,
  Active, // (A)
  Inactive, // (B)
} as const;

type ActivationType = PropertyValues<typeof Activation>;
type _ = Assert<Equal<
  ActivationType, typeof Active | typeof Inactive
>>;

type PropertyValues<Obj> = Obj[Exclude<keyof Obj, '__proto__'>];
```

Why the intermediate step of declaring the variables `Active` and `Inactive` in the first two lines? Why don't we create the symbols in line A and line B?

If we do that then:

- `Activation.Active` will have the type `symbol`, not the type `typeof Active`.
- `Activation.Inactive` will have the type `symbol`, not the type `typeof Inactive`.

As a result, `ActivationType` will be `symbol`. For a longer explanation, see "Symbols as property values".

16.5 Further reading

- TypeScript: chapter "Symbols" in the TypeScript Handbook
- JavaScript: chapter "Symbols" in "Exploring JavaScript"

Chapter 17

Adding special values to types

17.1 Adding special values in band . . . 139
17.1.1 Adding `null` or `undefined` to a type . . . 140
17.1.2 Adding a symbol to a type . . . 141
17.2 Adding special values out of band . . . 141
17.2.1 Discriminated unions . . . 142
17.2.2 Other kinds of union types . . . 143

One way of understanding types is as sets of values. Sometimes there are two levels of values:

- Base level: normal values
- Meta level: special values

In this chapter, we examine how we can add special values to base-level types.

17.1 Adding special values in band

One way of adding special values is to create a new type which is a superset of the base type where some values are special. These special values are called *sentinels*. They exist *in band* (think inside the same channel) and are siblings of normal values.

As an example, consider the following interface for a stream of text lines:

```
interface LineStream {
  getNextLine(): string;
}
```

At the moment, `.getNextLine()` only handles text lines, but not ends of files (EOFs). How could we add support for EOF?

Possibilities include:

- An additional method `.isEof()` that needs to be called before calling `.getNextLine()`.
- `.getNextLine()` throws an exception when it reaches an EOF.
- A sentinel value for EOF.

The next two subsections describe two ways in which we can introduce sentinel values.

17.1.1 Adding `null` or `undefined` to a type

When using strict TypeScript, no simple object type (defined via interfaces, object patterns, classes, etc.) includes `null`. That makes it a good sentinel value that we can add to the base type `string` via a union type:

```
type StreamValue = null | string;

interface LineStream {
  getNextLine(): StreamValue;
}
```

Now, whenever we are using the value returned by `.getNextLine()`, TypeScript forces us to consider both possibilities: strings and `null` – for example:

```
function countComments(ls: LineStream) {
  let commentCount = 0;
  while (true) {
    const line = ls.getNextLine();
    // @ts-expect-error: 'line' is possibly 'null'.
    if (line.startsWith('#')) { // (A)
      commentCount++;
    }
    if (line === null) break;
  }
  return commentCount;
}
```

In line A, we can't use the string method `.startsWith()` because `line` might be `null`. We can fix this as follows:

```
function countComments(ls: LineStream) {
  let commentCount = 0;
  while (true) {
    const line = ls.getNextLine();
    if (line === null) break;
    if (line.startsWith('#')) { // (A)
      commentCount++;
    }
  }
  return commentCount;
}
```

Now, when execution reaches line A, we can be sure that `line` is not `null`.

17.1.2 Adding a symbol to a type

We can also use values other than `null` as sentinels. Symbols are best suited for this task because each one of them has a unique identity and no other value can be mistaken for it.

This is how to use a symbol to represent EOF:

```
const EOF = Symbol('EOF');
type StreamValue = typeof EOF | string;
```

Why do we need `typeof` and can't use `EOF` directly? That's because `EOF` is a value, not a type. The type operator `typeof` converts `EOF` to a type.

Example: a symbol as an error value

The following function `parseNumber()` uses the symbol `couldNotParseNumber` as an error value:

```
const couldNotParseNumber = Symbol('couldNotParseNumber');

function parseNumber(str: string)
  : number | typeof couldNotParseNumber
{
  const result = Number(str);
  if (!Number.isNaN(result)) {
    return result;
  } else {
    return couldNotParseNumber;
  }
}

assert.equal(
  parseNumber('123'), 123
);
assert.equal(
  parseNumber('hello'), couldNotParseNumber
);
```

17.2 Adding special values out of band

What do we do if potentially *any* value can be returned by a method? How do we ensure that base values and meta values don't get mixed up? This is an example where that might happen:

```
interface ValueStream<T> {
  getNextValue(): T;
}
```

Whatever value we pick for `EOF`, there is a risk of someone creating an `ValueStream<typeof EOF>` and adding that value to the stream.

The solution is to keep normal values and special values separate, so that they can't be mixed up. Special values existing separately is called *out of band* (think different channel).

17.2.1 Discriminated unions

A *discriminated union* is a union of several object types that all have at least one property in common, the so-called *discriminant*. The discriminant must have a different value for each object type – we can think of it as the ID of the object type.

Example: `ValueStreamValue`

In the following example, `ValueStreamValue<T>` is a discriminated union and its discriminant is `.type`.

```
interface NormalValue<T> {
  type: 'normal'; // string literal type
  data: T;
}
interface Eof {
  type: 'eof'; // string literal type
}
type ValueStreamValue<T> = Eof | NormalValue<T>;

interface ValueStream<T> {
  getNextValue(): ValueStreamValue<T>;
}

function countValues<T>(vs: ValueStream<T>, data: T) {
  let valueCount = 0;
  while (true) {
    const value = vs.getNextValue(); // (A)
    assertType<Eof | NormalValue<T>>(value);

    if (value.type === 'eof') break;
    assertType<NormalValue<T>>(value); // (B)

    if (value.data === data) { // (C)
      valueCount++;
    }
  }
  return valueCount;
}
```

Initially, the type of `value` is `ValueStreamValue<T>` (line A). Then we exclude the value `'eof'` for the discriminant `.type` and its type is narrowed to `NormalValue<T>` (line B). That's why we can access property `.data` in line C.

Example: `IteratorResult`

When deciding how to implement iterators, TC39 didn't want to use a fixed sentinel value. Otherwise, code would break if that value appeared in an iterable. One solution would have been to pick a sentinel value when starting an iteration. TC39 instead opted for a discriminated union with the common property `.done`:

```
interface IteratorYieldResult<TYield> {
  done?: false; // boolean literal type
  value: TYield;
}

interface IteratorReturnResult<TReturn> {
  done: true; // boolean literal type
  value: TReturn;
}

type IteratorResult<T, TReturn = any> =
  | IteratorYieldResult<T>
  | IteratorReturnResult<TReturn>;
```

17.2.2 Other kinds of union types

Other kinds of union types can be as convenient as discriminated unions, as long as we have the means to distinguish the member types of the union.

One possibility is to distinguish the member types via unique properties:

```
interface A {
  one: number;
  two: number;
}
interface B {
  three: number;
  four: number;
}
type Union = A | B;

function func(x: Union) {
  // @ts-expect-error: Property 'two' does not exist on type 'Union'.
  // Property 'two' does not exist on type 'B'.(2339)
  console.log(x.two); // error

  if ('one' in x) { // discriminating check
    console.log(x.two); // OK
  }
}
```

Another possibility is to distinguish the member types via `typeof` and/or instance checks:

```
type Union = [string] | number;
```

```
function logHexValue(x: Union) {
  if (Array.isArray(x)) { // discriminating check
    console.log(x[0]); // OK
  } else {
    console.log(x.toString(16)); // OK
  }
}
```

Part V

Typing objects, classes and Arrays

Chapter 18

Typing objects

18.1 Object types 148
  18.1.1 The two ways of using objects 148
  18.1.2 Object types work structurally in TypeScript 149
18.2 Members of object literal types 149
  18.2.1 Method signatures 150
  18.2.2 Keys of object type members 151
  18.2.3 Modifiers of object type members 152
18.3 Excess property checks: When are extra properties allowed? 153
  18.3.1 Why are excess properties forbidden in object literals? 154
  18.3.2 Why are excess properties allowed if an object comes from somewhere else? 154
  18.3.3 Empty object literal types allow excess properties 155
  18.3.4 Matching only objects without properties 155
  18.3.5 Allowing excess properties in object literals 155
18.4 Object types and inherited properties 158
  18.4.1 TypeScript doesn't distinguish own and inherited properties 158
  18.4.2 Object literal types describe instances of `Object` 158
18.5 Interfaces vs. object literal types 158
  18.5.1 Object literal types can be inlined 159
  18.5.2 Interfaces with the same name are merged 159
  18.5.3 Mapped types look like object literal types 160
  18.5.4 Only interfaces support polymorphic `this` types 160
  18.5.5 Only interfaces support `extends` – but type intersection (`&`) is similar 160
18.6 Forbidding properties via `never` 163
  18.6.1 Forbidding properties with string keys 163
  18.6.2 Forbidding index properties (with number keys) 164
18.7 Index signatures: objects as dictionaries 164
  18.7.1 Typing index signature keys 165
  18.7.2 String keys vs. number keys 165
  18.7.3 Index signatures vs. property signatures and method signatures . . 166

18.8 Record<K, V> for dictionary objects . . . 167
18.8.1 Index signatures don't allow key unions . . . 167
18.8.2 Record enforces exhaustiveness for key unions . . . 167
18.8.3 Record: preventing exhaustiveness checks for key unions . . . 168
18.9 object vs Object vs. {} . . . 168
18.9.1 Plain JavaScript: objects vs. instances of Object . . . 169
18.9.2 Object (uppercase "O") in TypeScript: instances of class Object . . . 169
18.9.3 Type {} basically means: not nullish . . . 170
18.9.4 Inferred types for various objects . . . 171
18.10 Summary: object vs Object vs. {} vs. Record . . . 172
18.11 Sources of this chapter . . . 173

In this chapter, we will explore how objects and properties are typed statically in TypeScript.

18.1 Object types

18.1.1 The two ways of using objects

There are two ways of using objects in JavaScript:

- Fixed-layout object: Used this way, an object works like a record in a database. It has a fixed number of properties, whose keys are known at development time. Their values generally have different types.

  ```
  const fixedLayoutObject: FixedLayoutObjectType = {
    product: 'carrot',
    quantity: 4,
  };
  ```

- Dictionary object: Used this way, an object works like a lookup table or a map. It has a variable number of properties, whose keys are not known at development time. All of their values have the same type.

  ```
  const dictionaryObject: DictionaryObjectType = {
    ['one']: 1,
    ['two']: 2,
  };
  ```

Note that the two ways can also be mixed: Some objects are both fixed-layout objects and dictionary objects.

The most common ways of typing these two kinds of objects are:

```
type FixedLayoutObjectType = {
    product: string,
    quantity: number,
};
type DictionaryObjectType = Record<string, number>;
```

- `FixedLayoutObjectType` is an object literal type. The separators between properties can be either commas (`,`) or semicolons (`;`). I prefer the former because that's what JavaScript object literals use.
- `DictionaryObjectType` uses the utility type `Record` to define a type for dictionary objects whose keys are strings and whose values are numbers.

Next, we'll look at fixed-layout object types in more detail before coming back to dictionary object types.

18.1.2 Object types work structurally in TypeScript

Object types work structurally in TypeScript: They match all values that have their structure. Therefore, a type can be defined after a given value and still match it – e.g.:

```
const myPoint = {x: 1, y: 2};

function logPoint(point: {x: number, y: number}): void {
  console.log(point);
}

logPoint(myPoint); // Works!
```

For more information on this topic, see "Nominal type systems vs. structural type systems" (§13.4).

18.2 Members of object literal types

The constructs inside the bodies of object literal types are called their *members*. These are the most common members:

```
type ExampleObjectType = {
  // Property signature
  myProperty: boolean,

  // Method signature
  myMethod(str: string): number,

  // Index signature
  [key: string]: any,

  // Call signature
  (num: number): string,

  // Construct signature
  new(str: string): ExampleInstanceType,
};

type ExampleInstanceType = {};
```

Let's look at these members in more detail:

- Property signatures define properties and should be self-explanatory:

  ```
  myProperty: boolean;
  ```

- Method signatures define methods and are described in the next subsection.

  ```
  myMethod(str: string): number;
  ```

  Note: The names of parameters (in this case: `str`) help with documenting how things work but have no other purpose.

- Index signatures are needed to describe Arrays or objects that are used as dictionaries. They are described later in this chapter.

  ```
  [key: string]: any;
  ```

  Note: The name `key` is only there for documentation purposes.

- Call signatures enable object literal types to describe functions. See "Interfaces with call signatures".

  ```
  (num: number): string;
  ```

- Construct signatures enable object literal types to describe classes and constructor functions. See "Object type literals with construct signatures" (§23.2.3).

  ```
  new(str: string): ExampleInstanceType;
  ```

18.2.1 Method signatures

As far as TypeScript's type system is concerned, method definitions and properties whose values are functions, are equivalent:

```
type HasMethodDef = {
  simpleMethod(flag: boolean): void,
};
type HasFuncProp = {
  simpleMethod: (flag: boolean) => void,
};
type _ = Assert<Equal<
  HasMethodDef,
  HasFuncProp
>>;

const objWithMethod = {
  simpleMethod(flag: boolean): void {},
};
assertType<HasMethodDef>(objWithMethod);
assertType<HasFuncProp>(objWithMethod);

const objWithOrdinaryFunction: HasMethodDef = {
  simpleMethod: function (flag: boolean): void {},
};
assertType<HasMethodDef>(objWithOrdinaryFunction);
```

```
assertType<HasFuncProp>(objWithOrdinaryFunction);

const objWithArrowFunction: HasMethodDef = {
  simpleMethod: (flag: boolean): void => {},
};
assertType<HasMethodDef>(objWithArrowFunction);
assertType<HasFuncProp>(objWithArrowFunction);
```

My recommendation is to use whichever syntax best expresses how a property should be set up.

18.2.2 Keys of object type members

Quoted keys

Just like in JavaScript, property keys can be quoted:

```
type Obj = { 'hello everyone!': string };
```

Unquoted numbers as keys

This rarely matters in practice, but as an aside: Just like in JavaScript, we can use unquoted numbers as keys. Unlike JavaScript, those keys are considered to be number literal types:

```
type _ = Assert<Equal<
  keyof {0: 'a', 1: 'b'},
  0 | 1
>>;
```

For comparison, this is how JavaScript works:

```
assert.deepEqual(
  Object.keys({0: 'a', 1: 'b'}),
  [ '0', '1' ]
);
```

For more information see $type.

Computed property keys

Computed property keys are a JavaScript feature. There is a similar feature at the type level:

```
type ExampleObjectType = {
  // Property signature with computed key
  [Symbol.toStringTag]: string,

  // Method signature with computed key
  [Symbol.iterator](): IteratorObject<string>,
};
```

Unexpectedly, computed property keys are values, not types. TypeScript internally applies `typeof` to create the type:

```
type _ = Assert<Equal<
  { ['hello']: string },
  { hello: string }
>>;
```

What kind of value is allowed as a computed property key? Its type must be:

- A string literal type such as `'abc'`
- A number literal type such as `123`
- A unique symbol type
- `any`

18.2.3 Modifiers of object type members

Optional properties

If we put a question mark (`?`) after the name of a property, that property is optional. The same syntax is used to mark parameters of functions, methods, and constructors as optional. In the following example, property `.middle` is optional:

```
type Name = {
  first: string;
  middle?: string;
  last: string;
};
```

Therefore, it's OK to omit that property (line A):

```
const john: Name = {first: 'Doe', last: 'Doe'}; // (A)
const jane: Name = {first: 'Jane', middle: 'Cecily', last: 'Doe'};
```

Optional vs. `undefined | string` with `exactOptionalPropertyTypes` In this book, all code uses the compiler setting `exactOptionalPropertyTypes`. With that setting, the difference an optional property and a property with type `undefined | string` is intuitive:

```
type Obj = {
  prop1?: string;
  prop2: undefined | string;
};

const obj1: Obj = { prop1: 'a', prop2: 'b' };

// .prop1 can be omitted; .prop2 can be `undefined`
const obj2: Obj = { prop2: undefined };

// @ts-expect-error: Type '{ prop1: undefined; prop2: string; }' is not
// assignable to type 'Obj' with 'exactOptionalPropertyTypes: true'.
// Consider adding 'undefined' to the types of the target's properties.
// Types of property 'prop1' are incompatible. Type 'undefined' is not
// assignable to type 'string'.
const obj3: Obj = { prop1: undefined, prop2: 'b' };
```

```
// @ts-expect-error: Property 'prop2' is missing in type '{ prop1: string;
// }' but required in type 'Obj'.
const obj4: Obj = { prop1: 'a' };
```

Types such as `undefined | string` and `null | string` are useful if we want to make omissions explicit. When people see such an explicitly omitted property, they know that it exists but was switched off.

Optional vs. `undefined | string` without `exactOptionalPropertyTypes` If `exactOptionalPropertyTypes` is `false` then one thing changes: `.prop1` can also be `undefined`:

```
type Obj = {
  prop1?: string;
  prop2: undefined | string;
};

const obj1: Obj = { prop1: undefined, prop2: undefined };
```

Read-only properties

In the following example, property `.prop` is read-only:

```
type MyObj = {
  readonly prop: number;
};
```

As a consequence, we can read it, but we can't change it:

```
const obj: MyObj = {
  prop: 1,
};

console.log(obj.prop); // OK

// @ts-expect-error: Cannot assign to 'prop' because it is a read-only
// property.
obj.prop = 2;
```

18.3 Excess property checks: When are extra properties allowed?

As an example, consider the following object literal type:

```
type Point = {
  x: number,
  y: number,
};
```

There are two ways (among others) in which this object literal type could be interpreted:

- Closed interpretation: It could describe all objects that have *exactly* the properties `.x` and `.y` with the specified types. On other words: Those objects must not have *excess properties* (more than the required properties).
- Open interpretation: It could describe all objects that have *at least* the properties `.x` and `.y`. In other words: Excess properties are allowed.

TypeScript uses both interpretations. To explore how that works, we will use the following function:

```
function computeDistance(point: Point) { /*...*/ }
```

The default is that the excess property `.z` is allowed:

```
const obj = { x: 1, y: 2, z: 3 };
computeDistance(obj); // OK
```

However, if we use object literals directly, then excess properties are forbidden:

```
// @ts-expect-error: Object literal may only specify known properties, and
// 'z' does not exist in type 'Point'.
computeDistance({ x: 1, y: 2, z: 3 }); // error

computeDistance({x: 1, y: 2}); // OK
```

18.3.1 Why are excess properties forbidden in object literals?

Why the stricter rules for object literals? They provide protection against typos in property keys. We will use the following object literal type to demonstrate what that means.

```
type Person = {
  first: string,
  middle?: string,
  last: string,
};
function computeFullName(person: Person) { /*...*/ }
```

Property `.middle` is optional and can be omitted. To TypeScript, mistyping its name looks like omitting it and providing an excess property. However, it still catches the typo because excess properties are not allowed in this case:

```
// @ts-expect-error: Object literal may only specify known properties, but
// 'mdidle' does not exist in type 'Person'. Did you mean to write
// 'middle'?
computeFullName({first: 'Jane', mdidle: 'Cecily', last: 'Doe'});
```

18.3.2 Why are excess properties allowed if an object comes from somewhere else?

The idea is that if an object comes from somewhere else, we can assume that it has already been vetted and will not have any typos. Then we can afford to be less careful.

If typos are not an issue, our goal should be maximizing flexibility. Consider the following function:

```
type HasYear = {
  year: number,
};

function getAge(obj: HasYear) {
  const yearNow = new Date().getFullYear();
  return yearNow - obj.year;
}
```

Without allowing excess properties for values that are passed to `getAge()`, the usefulness of this function would be quite limited.

18.3.3 Empty object literal types allow excess properties

If an object literal type is empty, excess properties are always allowed:

```
type Empty = {};
type OneProp = {
  myProp: number,
};

// @ts-expect-error: Object literal may only specify known properties, and
// 'anotherProp' does not exist in type 'OneProp'.
const a: OneProp = { myProp: 1, anotherProp: 2 };
const b: Empty = { myProp: 1, anotherProp: 2 }; // OK
```

18.3.4 Matching only objects without properties

If we want to enforce that an object has no properties, we can use the following trick (credit: Geoff Goodman):

```
type WithoutProperties = {
  [key: string]: never,
};

// @ts-expect-error: Type 'number' is not assignable to type 'never'.
const a: WithoutProperties = { prop: 1 };
const b: WithoutProperties = {}; // OK
```

18.3.5 Allowing excess properties in object literals

What if we want to allow excess properties in object literals? As an example, consider type `Point` and function `computeDistance1()`:

```
type Point = {
  x: number,
  y: number,
};

function computeDistance1(point: Point) { /*...*/ }
```

```
// @ts-expect-error: Object literal may only specify known properties, and
// 'z' does not exist in type 'Point'.
computeDistance1({ x: 1, y: 2, z: 3 });
```

One option is to assign the object literal to an intermediate variable:

```
const obj = { x: 1, y: 2, z: 3 };
computeDistance1(obj);
```

A second option is to use a type assertion:

```
computeDistance1({ x: 1, y: 2, z: 3 } as Point); // OK
```

A third option is to rewrite `computeDistance1()` so that it uses a type parameter:

```
function computeDistance2<P extends Point>(point: P) { /*...*/ }
computeDistance2({ x: 1, y: 2, z: 3 }); // OK
```

A fourth option is to extend `Point` so that it allows excess properties:

```
type PointEtc = Point & {
  [key: string]: any;
};
function computeDistance3(point: PointEtc) { /*...*/ }

computeDistance3({ x: 1, y: 2, z: 3 }); // OK
```

We used an intersection type (`&` operator) to define `PointEtc`. For more information, see "Intersections of object types" (§20.1).

We'll continue with two examples where TypeScript not allowing excess properties, is a problem.

Allowing excess properties: example `Incrementor` factory

In this example, we implement a factory for objects of type `Incrementor` and would like to return a subtype, but TypeScript doesn't allow the extra property `.counter`:

```
type Incrementor = {
  inc(): number,
};
function createIncrementor(): Incrementor {
  return {
    // @ts-expect-error: Object literal may only specify known properties, and
    // 'counter' does not exist in type 'Incrementor'.
    counter: 0,
    inc() {
      // @ts-expect-error: Property 'counter' does not exist on type
      // 'Incrementor'.
      return this.counter++;
    },
  };
}
```

Alas, even with a type assertion, there is still one type error:

```
function createIncrementor2(): Incrementor {
  return {
    counter: 0,
    inc() {
      // @ts-expect-error: Property 'counter' does not exist on type
      // 'Incrementor'.
      return this.counter++;
    },
  } as Incrementor;
}
```

What does work is `as any` but then the type of the returned object is `any` and, e.g. inside `.inc()`, TypeScript doesn't check if properties of `this` really exist.

A proper solution is to add an index signature to `Incrementor`. Or – especially if that is not possible – to introduce an intermediate variable:

```
function createIncrementor3(): Incrementor {
  const incrementor = {
    counter: 0,
    inc() {
      return this.counter++;
    },
  };
  return incrementor;
}
```

Allowing excess properties: example `.dateStr`

The following comparison function can be used to sort objects that have the property `.dateStr`:

```
function compareDateStrings(
  a: {dateStr: string}, b: {dateStr: string}) {
    if (a.dateStr < b.dateStr) {
      return +1;
    } else if (a.dateStr > b.dateStr) {
      return -1;
    } else {
      return 0;
    }
  }
```

For example in unit tests, we may want to invoke this function directly with object literals. TypeScript doesn't let us do this and we need to use one of the workarounds.

18.4 Object types and inherited properties

18.4.1 TypeScript doesn't distinguish own and inherited properties

TypeScript doesn't distinguish own and inherited properties. They are all simply considered to be properties.

```
type MyType = {
  toString(): string, // inherited property
  prop: number, // own property
};
const obj: MyType = { // OK
  prop: 123,
};
```

`obj` inherits `.toString()` from `Object.prototype`.

The downside of this approach is that some phenomena in JavaScript can't be described via TypeScript's type system. The upside is that the type system is simpler.

18.4.2 Object literal types describe instances of `Object`

All object literal types describe objects that are instances of `Object` and inherit the properties of `Object.prototype`. In the following example, the parameter `x` of type `{}` is compatible with the return type `Object`:

```
function f1(x: {}): Object {
  return x;
}
```

Similarly, `{}` has a method `.toString()`:

```
function f2(x: {}): { toString(): string } {
  return x;
}
```

18.5 Interfaces vs. object literal types

For historical reasons, object types can be defined in two ways:

```
// Object literal type
type ObjType1 = {
  a: boolean,
  b: number,
  c: string,
};

// Interface
interface ObjType2 {
  a: boolean;
  b: number;
```

```
  c: string;
}
```

- In both cases, either semicolons or commas can be used as separators. I prefer commas for object literal types and semicolons for interfaces because that reflects what JavaScript looks like (object literals and classes).
- Trailing separators are allowed and optional.

Both ways of defining an object type are more or less equivalent now. We'll dive into the (minor) differences next.

18.5.1 Object literal types can be inlined

Object literal types can be inlined, while interfaces can't be:

```
// The object literal type is inlined
// (mentioned inside the parameter definition)
function f1(x: {prop: number}) {}

// We can't mention the interface inside the parameter definition.
// We can only define it externally and refer to it.
function f2(x: ObjectInterface) {}
interface ObjectInterface {
  prop: number;
}
```

18.5.2 Interfaces with the same name are merged

Type aliases with duplicate names are illegal:

```
// @ts-expect-error: Duplicate identifier 'PersonAlias'.
type PersonAlias = {first: string};
// @ts-expect-error: Duplicate identifier 'PersonAlias'.
type PersonAlias = {last: string};
```

Conversely, interfaces with duplicate names are merged:

```
interface PersonInterface {
  first: string;
}
interface PersonInterface {
  last: string;
}
const jane: PersonInterface = {
  first: 'Jane',
  last: 'Doe',
};
```

This is called declaration merging and can be used to combine types from multiple sources – e.g., as long as `Array.fromAsync()` is a new method, it is not part of the core library declaration file, but provided via `lib.esnext.array.d.ts` – which adds it as an increment to `ArrayConstructor` (the type of `Array` as a class value):

```
interface ArrayConstructor {
  fromAsync<T>(···): Promise<T[]>;
}
```

18.5.3 Mapped types look like object literal types

A mapped type (line A) looks like an object literal type:

```
type Point = {
  x: number,
  y: number,
};

type PointCopy1 = {
  [Key in keyof Point]: Point[Key] // (A)
};
```

As an option, we can end line A with a semicolon. Alas, a comma is not allowed.

For more information on this topic, see “Mapped types `{[K in U]: X}`”.

18.5.4 Only interfaces support polymorphic `this` types

Polymorphic `this` types can only be used in interfaces:

```
interface AddsStrings {
  add(str: string): this;
};

class StringBuilder implements AddsStrings {
  result = '';
  add(str: string): this {
    this.result += str;
    return this;
  }
}
```

18.5.5 Only interfaces support `extends` – but type intersection (`&`) is similar

An interface `B` can extend another interface `A` and is then interpreted as an increment of `A`:

```
interface A {
  propA: number;
}
interface B extends A {
  propB: number;
}
type _ = Assert<Equal<
  B,
```

```
    {
      propA: number,
      propB: number,
    }
  >>;
```

Object literal types don't support `extend` but an intersection type `&` has a similar effect:

```
type A = {
  propA: number,
};
type B = {
  propB: number,
} & A;
type _ = Assert<Equal<
  B,
  {
    propA: number,
    propB: number,
  }
>>;
```

Intersections of object types are described in more detail in another chapter. Here, we'll explore how exactly they differ from `extends`.

Conflicts

If there is a conflict between an extending interface and an extended interface then that's an error:

```
interface A {
  prop: string;
}
// @ts-expect-error: Interface 'B' incorrectly extends interface 'A'.
// Types of property 'prop' are incompatible.
interface B extends A {
  prop: number;
}
```

In contrast, intersection types don't complain about conflicts, but they may result in `never` in some locations:

```
type A = {
  prop: string,
};
type B = {
  prop: number,
} & A;
type _ = Assert<Equal<
  B,
  {
```

```
    prop: number & string, // never
  }
>>;
```

Only interfaces support overriding

Overriding a method means replacing a method in a supertype with a *compatible* method – roughly:

- The overriding method can return more specific values – e.g. invokers of the overridden method that expect an `Object` won't mind if the overriding method returns a `RegExp`.
- The overriding method can expect less specific parameters – e.g. invokers of the overridden method that pass an argument of type `string` won't mind if the overriding method accepts `string | number`.

```
interface A {
  m(x: string): Object;
}
interface B extends A {
  m(x: string | number): RegExp;
}

type _ = Assert<Equal<
  B,
  {
    m(x: string | number): RegExp,
  }
>>;

function f(x: B) {
  assertType<RegExp>(x.m('abc'));
}
```

We can see that the overriding method "wins" and completely replaces the overridden method in `B`. In contrast, both methods exist in parallel in an intersection type:

```
type A = {
  m(x: string): Object,
};
type B = {
  m(x: string | number): RegExp,
};

type _ = [
  Assert<Equal<
    A & B,
    {
      m: ((x: string) => Object) & ((x: string | number) => RegExp),
    }
```

```
    >>,
    Assert<Equal<
      B & A,
      {
        m: ((x: string | number) => RegExp) & ((x: string) => Object),
      }
    >>,
  ];

  function f1(x: A & B) {
    assertType<Object>(x.m('abc')); // (A)
  }
  function f2(x: B & A) {
    assertType<RegExp>(x.m('abc')); // (B)
  }
```

When it comes to the return type (line A and line B), the earlier member of the intersection wins. That's why `B & A` (`B1`) is more similar to `B extends A`, even though `A & B` (`B2`) looks nicer:

```
type B1 = {
  prop: number,
} & A;
type B2 = A & {
  prop: number,
};
```

`extends` or `&` – which one to use?

Which one to use depends on the context. If inheritance is involved then an interface and `extends` is usually the better choice due to their support of overriding.

Source of this section

- GitHub issue "TypeScript: types vs. interfaces" by Johannes Ewald

18.6 Forbidding properties via `never`

Given that no other type is assignable to `never`, we can use it to forbid properties.

18.6.1 Forbidding properties with string keys

The type `EmptyObject` forbids string keys:

```
type EmptyObject = Record<string, never>;

// @ts-expect-error: Type 'number' is not assignable to type 'never'.
```

```
const obj1: EmptyObject = { prop: 123 };
const obj2: EmptyObject = {}; // OK
```

In contrast, the type `{}` is assignable from all objects and not a type for empty objects:

```
const obj3: {} = { prop: 123 };
```

18.6.2 Forbidding index properties (with number keys)

The type `NoIndices` forbids number keys but allows the string key `'prop'`:

```
type NoIndices = Record<number, never> & { prop?: boolean };

//===== Objects =====
const obj1: NoIndices = {}; // OK
const obj2: NoIndices = { prop: true }; // OK
// @ts-expect-error: Type 'string' is not assignable to type 'never'.
const obj3: NoIndices = { 0: 'a' }; // OK

//===== Arrays =====
const arr1: NoIndices = []; // OK
// @ts-expect-error: Type 'string' is not assignable to type 'never'.
const arr2: NoIndices = ['a'];
```

18.7 Index signatures: objects as dictionaries

So far, we have only used types for fixed-layout objects. How do we express the fact that an object is to be used as a dictionary? For example: What should `TranslationDict` be in the following code fragment?

```
function translate(dict: TranslationDict, english: string): string {
  const translation = dict[english];
  if (translation === undefined) {
    throw new Error();
  }
  return translation;
}
```

One option is to use an index signature (line A) to express that `TranslationDict` is for objects that map string keys to string values (another option is `Record` – which we'll get to later):

```
type TranslationDict = {
  [key: string]: string, // (A)
};
const dict = {
  'yes': 'sí',
  'no': 'no',
  'maybe': 'tal vez',
};
assert.equal(
```

```
translate(dict, 'maybe'),
'tal vez');
```

The name key doesn't matter – it can be any identifier and is ignored (but can't be omitted).

18.7.1 Typing index signature keys

An index signature represents an infinite set of properties; only the following types are allowed:

- `string`
- `number`
- `symbol`
- A template string literal with an infinite primitive type – e.g.: `` `${bigint}` ``
- A union of any of the previous types

Specifically *not* allowed are:

- A single string literal type – e.g.: `'a'`, `1`, `false`
- A union of string literal types – e.g.:
  - `'a' | 'b'`
  - `1 | 2`
  - `boolean` (which is `false | true`)
- A template string literal with one of the previous types – e.g.: `` `${boolean}` ``
- The types: `never`, `any`, `unknown`

If you need more power then consider using a mapped types.

These are examples of index signatures:

```
type IndexSignature1 = {
  [key: string]: boolean,
};
// Template string literal with infinite primitive type
type IndexSignature2 = {
  [key: `${bigint}`]: string,
};
// Union of previous types
type IndexSignature3 = {
  [key: string | `${bigint}`]: string,
};
```

18.7.2 String keys vs. number keys

Just like in plain JavaScript, TypeScript's number property keys are a subset of the string property keys (see "Exploring JavaScript"). Accordingly, if we have both a string index signature and a number index signature, the property type of the former must be a supertype of the latter. The following example works because `Object` is a supertype of `RegExp` (`RegExp` is assignable to `Object`):

```
type StringAndNumberKeys = {
  [key: string]: Object,
```

```
  [key: number]: RegExp,
};
```

The following code demonstrates the effects of using strings and numbers as property keys:

```
function f(x: StringAndNumberKeys) {
  return {
    str: x['abc'],
    num: x[123],
  };
}
assertType<
  (x: StringAndNumberKeys) => {
    str: Object | undefined,
    num: RegExp | undefined,
  }
>(f);
```

18.7.3 Index signatures vs. property signatures and method signatures

If there are both an index signature and property and/or method signatures in an object literal type, then the type of the index property value must also be a supertype of the type of the property value and/or method.

```
type T1 = {
  [key: string]: boolean,

  // @ts-expect-error: Property 'myProp' of type 'number' is not assignable
  // to 'string' index type 'boolean'.
  myProp: number,

  // @ts-expect-error: Property 'myMethod' of type '() => string' is not
  // assignable to 'string' index type 'boolean'.
  myMethod(): string,
};
```

In contrast, the following two object literal types produce no errors:

```
type T2 = {
  [key: string]: number,
  myProp: number,
};

type T3 = {
  [key: string]: () => string,
  myMethod(): string,
}
```

18.8 Record<K, V> for dictionary objects

The built-in generic utility type `Record<K, V>` is for dictionary objects whose keys are of type `K` and whose values are of type `V`:

```
const dict: Record<string, number> = {
  one: 1,
  two: 2,
  three: 3,
};
```

If you are curious how `Record` is defined: "`Record` is a mapped type". This knowledge can help with remembering how it handles finite and infinite key types.

`Record` supports unions of literal types as key types; index signatures don't. More on that next.

18.8.1 Index signatures don't allow key unions

The key type of an index signature must be infinite:

```
type Key = 'A' | 'B' | 'C';

// @ts-expect-error: An index signature parameter type cannot be a literal
// type or generic type. Consider using a mapped object type instead.
const dict: {[key: Key]: true} = {
  A: true,
  C: true,
};
```

18.8.2 Record enforces exhaustiveness for key unions

`Record` enforces exhaustiveness if its key type is a union of literal types:

```
type T = 'A' | 'B' | 'C';

// @ts-expect-error: Property 'C' is missing in type '{ A: true; B: true; }'
// but required in type 'Record<T, true>'.
const nonExhaustiveKeys: Record<T, true> = {
  A: true,
  B: true,
};

const exhaustiveKeys: Record<T, true> = {
  A: true,
  B: true,
  C: true,
};
```

Wrong keys also produce errors:

```
const wrongKey: Record<T, true> = {
  A: true,
  B: true,
  // @ts-expect-error: Object literal may only specify known properties,
  // and 'D' does not exist in type 'Record<T, true>'.
  D: true,
};
```

18.8.3 Record: preventing exhaustiveness checks for key unions

If we want to prevent exhaustiveness checks for keys whose type is a union then we can use the utility type `Partial` (which makes all properties optional). Then we can omit some properties, but wrong keys still produce errors:

```
type T = 'A' | 'B' | 'C';
const nonExhaustiveKeys: Partial<Record<T, true>> = {
  A: true,
};
const wrongKey: Partial<Record<T, true>> = {
  // @ts-expect-error: Object literal may only specify known properties,
  // and 'D' does not exist in type 'Partial<Record<T, true>>'.
  D: true,
};
```

18.9 object vs Object vs. {}

These are three similar general types for objects:

- `object` with a lowercase "o" is the type of all non-primitive values. It's loosely related to the value `'object'` returned by the JavaScript operator `typeof`.

  ```
  const obj1: object = {};
  const obj2: object = [];
  // @ts-expect-error: Type 'number' is not assignable to type 'object'.
  const obj3: object = 123;
  ```

- `Object` with an uppercase "O" is the type of the instances of class `Object`:

  ```
  const obj1: Object = new Object();
  ```

  But it also accepts primitive values (except for `undefined` and `null`):

  ```
  const obj2: Object = 123;
  ```

  Note that non-nullish primitive values inherit the methods of `Object.prototype` via their wrapper types.

- `{}` accepts all non-nullish values. Its only difference with `Object` is that it doesn't mind if a property conflicts with `Object.prototype` properties:

  ```
  const obj1: {} = { toString: true }; // OK
  const obj2: Object = {
    // @ts-expect-error: Type 'boolean' is not assignable to
  ```

```
    // type '() => string'.
    toString: true,
  };
```

So the type {} basically means: "Value must not be null".

These types are not used that often

Given that we can't access any properties if we use these types, they are not used that often. If a value does have that type, we usually narrow its type via a type guard before doing anything with it.

18.9.1 Plain JavaScript: objects vs. instances of `Object`

In plain JavaScript, there is an important distinction.

On one hand, most objects are *instances* of `Object`.

```
> const obj1 = {};
> obj1 instanceof Object
true
```

That means:

- `Object.prototype` is in their prototype chains (that's what `instanceof` checks):

  ```
  > Object.prototype.isPrototypeOf(obj1)
  true
  ```

- They inherit its properties.

  ```
  > obj1.toString === Object.prototype.toString
  true
  ```

On the other hand, we can also create objects that don't have `Object.prototype` in their prototype chains. For example, the following object does not have any prototype at all:

```
> const obj2 = Object.create(null);
> Object.getPrototypeOf(obj2)
null
```

`obj2` is an object that is not an instance of class `Object`:

```
> typeof obj2
'object'
> obj2 instanceof Object
false
```

18.9.2 `Object` (uppercase "O") in TypeScript: instances of class `Object`

Recall that each class `C` creates two entities:

- A constructor function `C`.

- An object type `C` that describes instances of the constructor function.

Similarly, there are two object types for class `Object`:

- Type `Object` specifies the properties of instances of `Object`, including the properties inherited from `Object.prototype`.
- Type `ObjectConstructor` specifies the properties of class `Object` (an object with properties).

These are the types:

```
interface Object { // (A)
  constructor: Function;
  toString(): string;
  toLocaleString(): string;
  /** Returns the primitive value of the specified object. */
  valueOf(): Object; // (B)
  hasOwnProperty(v: PropertyKey): boolean;
  isPrototypeOf(v: Object): boolean;
  propertyIsEnumerable(v: PropertyKey): boolean;
}

interface ObjectConstructor {
  /** Invocation via `new` */
  new(value?: any): Object;
  /** Invocation via function calls */
  (value?: any): any;

  readonly prototype: Object; // (C)

  getPrototypeOf(o: any): any;
  // ···
}
declare var Object: ObjectConstructor; // (D)
```

Observations:

- We have both a variable whose name is `Object` (line D) and a type whose name is `Object` (line A).
- `Object.prototype` also has the type `Object` (line C). Given that any instance of `Object` inherits all of its properties, that makes sense.
- It's interesting that, in line B, `.valueOf()` has the return type `Object` and is supposed to return primitive values.

18.9.3 Type {} basically means: not nullish

`{}` accepts all values other than `undefined` and `null`:

```
const v1: {} = 123;
const v2: {} = 123;
const v3: {} = {};
```

```
const v4: {} = { prop: true };

// @ts-expect-error: Type 'undefined' is not assignable to type '{}'.
const v5: {} = undefined;
// @ts-expect-error: Type 'null' is not assignable to type '{}'.
const v6: {} = null;
```

The helper type `NonNullable` uses `{}`:

```
/**
 * Exclude null and undefined from T
 */
type NonNullable<T> = T & {};

type _ = [
  Assert<Equal<
    NonNullable<undefined | string>,
    string
  >>,
  Assert<Equal<
    NonNullable<null | string>,
    string
  >>,
  Assert<Equal<
    NonNullable<string>,
    string
  >>,
];
```

The result of `NonNullable<T>` is a type that is the intersection of `T` and all non-nullish values.

18.9.4 Inferred types for various objects

These are the types that TypeScript infers for objects that are created via various means:

```
const obj1 = new Object();
assertType<Object>(obj1);

const obj2 = Object.create(null);
assertType<any>(obj2);

const obj3 = {};
assertType<{}>(obj3);

const obj4 = {prop: 123};
assertType<{prop: number}>(obj4);

const obj5 = Reflect.getPrototypeOf({});
assertType<object | null>(obj5);
```

In principle, the return type of `Object.create()` could (and probably should) be `object`

or a computed type. However, for historic reasons, it is `any`. That allows us to add and change properties of the result.

18.10 Summary: `object` vs `Object` vs. `{}` vs. `Record`

The following table compares four types for objects:

	`object`	`Object`	`{}`	`Record`
Accepts `undefined` or `null`	✘	✘	✘	✘
Accepts primitive values	✘	✔	✔	✘
Has `.toString()`	✔	✔	✔	N/A
Values can conflict with `Object`	✔	✘	✔	N/A

The last two table rows don't really make sense for `Record` – which is why there is an "N/A" in its cells.

Accepts `undefined` or `null`:

```
type _ = [
  Assert<Not<Assignable<
    object, undefined
  >>>,
  Assert<Not<Assignable<
    Object, undefined
  >>>,
  Assert<Not<Assignable<
    {}, undefined
  >>>,
  Assert<Not<Assignable<
    Record<keyof any, any>, undefined
  >>>,
];
```

Accepts primitive values:

```
type _ = [
  Assert<Not<Assignable<
    object, 123
  >>>,
  Assert<Assignable<
    Object, 123
  >>,
  Assert<Assignable<
    {}, 123
  >>,
  Assert<Not<Assignable<
    Record<keyof any, any>, 123
  >>>,
];
```

Has `.toString()`:

```
type _ = [
  Assert<Assignable<
    { toString(): string }, object
  >>,
  Assert<Assignable<
    { toString(): string }, Object
  >>,
  Assert<Assignable<
    { toString(): string }, {}
  >>,
];
```

Values can conflict with `Object`:

```
type _ = [
  Assert<Assignable<
    object, { toString(): number }
  >>,
  Assert<Not<Assignable<
    Object, { toString(): number }
  >>>,
  Assert<Assignable<
    {}, { toString(): number }
  >>,
];
```

18.11 Sources of this chapter

- TypeScript Handbook

Chapter 19

Unions of object types

19.1 From unions of object types to discriminated unions . . . 175
19.1.1 Example: a union of objects . . . 176
19.1.2 `FileEntry` as a discriminated union . . . 177
19.1.3 Discriminated unions are related to algebraic data types . . . 178
19.1.4 `readFile()` for the new `FileEnty` . . . 178
19.1.5 Pros and cons of discriminated unions . . . 179
19.2 Deriving types from discriminated unions . . . 180
19.2.1 Extracting the values of the discriminant (the type tags) . . . 181
19.2.2 Maps for the elements of discriminated unions . . . 181
19.2.3 Extracting a subtype of a discriminated union . . . 182
19.3 Class hierarchies vs. discriminated unions . . . 183
19.3.1 A class hierarchy for syntax trees . . . 183
19.3.2 A discriminated union for syntax trees . . . 184
19.3.3 Comparing classes and discriminated unions . . . 185
19.4 Defining discriminated unions via classes . . . 185

In this chapter, we explore what unions of object types can be used for in TypeScript.

In this chapter, *object type* means:

- Object literal type
- Interface type
- Mapped type (such as `Record`)

19.1 From unions of object types to discriminated unions

Unions of object types are often a good choice if a single type has multiple representations – e.g. a type `Shape` that can be either a `Triangle`, a `Rectangle` or a `Circle`:

```
type Shape = Triangle | Rectangle | Circle;

type Triangle = {
  corner1: Point,
  corner2: Point,
  corner3: Point,
};
type Rectangle = {
  corner1: Point,
  corner2: Point,
};
type Circle = {
  center: Point,
  radius: number,
};

type Point = {
  x: number,
  y: number,
};
```

19.1.1 Example: a union of objects

The following types define a simple virtual file system:

```
type VirtualFileSystem = Map<string, FileEntry>;

type FileEntry = FileEntryData | FileEntryGenerator | FileEntryFile;
type FileEntryData = {
  data: string,
};
type FileEntryGenerator = {
  generator: (path: string) => string,
};
type FileEntryFile = {
  path: string,
};
```

A function `readFile()` for `VirtualFileSystem` would work as follows (line A and line B):

```
const vfs: VirtualFileSystem = new Map([
  [ '/tmp/file.txt',
    { data: 'Hello!' }
  ],
  [ '/tmp/echo.txt',
    { generator: (path: string) => path }
  ],
]);
assert.equal(
  readFile(vfs, '/tmp/file.txt'), // (A)
```

```
    'Hello!'
  );
  assert.equal(
    readFile(vfs, '/tmp/echo.txt'), // (B)
    '/tmp/echo.txt'
  );
```

This is an implementation of `readFile()`:

```
import * as fs from 'node:fs';
function readFile(vfs: VirtualFileSystem, path: string): string {
  const fileEntry = vfs.get(path);
  if (fileEntry === undefined) {
    throw new Error('Unknown path: ' + JSON.stringify(path));
  }
  if ('data' in fileEntry) { // (A)
    return fileEntry.data;
  } else if ('generator' in fileEntry) { // (B)
    return fileEntry.generator(path);
  } else if ('path' in fileEntry) { // (C)
    return fs.readFileSync(fileEntry.path, 'utf-8');
  } else {
    throw new UnexpectedValueError(fileEntry); // (D)
  }
}
```

Initially, the type of `fileEntry` is `FileEntry` and therefore:

```
FileEntryData | FileEntryGenerator | FileEntryFile
```

We have to narrow its type to one of the elements of this union type before we can access properties. And TypeScript lets us do that via the `in` operator (line A, line B, line C).

Additionally, we check statically if we covered all possible cases, by checking if `fileEntry` is assignable to the type `never` (line D). That is done via the following class:

```
class UnexpectedValueError extends Error {
  constructor(_value: never) {
    super();
  }
}
```

For more information on this technique and a longer and better implementation of `UnexpectedValueError`, see "Use case for `never`: exhaustiveness checks at compile time" (§15.4).

19.1.2 `FileEntry` as a discriminated union

A *discriminated union* is a union of object types that all have one property in common – whose value indicates the type of a union element. Let's convert `FileEntry` to a discriminated union:

```
type FileEntry =
  | {
    kind: 'FileEntryData',
    data: string,
  }
  | {
    kind: 'FileEntryGenerator',
    generator: (path: string) => string,
  }
  | {
    kind: 'FileEntryFile',
    path: string,
  }
  ;
type VirtualFileSystem = Map<string, FileEntry>;
```

The property of a discriminated union that has the type information is called a *discriminant* or a *type tag*. The discriminant of `FileEntry` is `.kind`. Other common names are `.tag`, `.key` and `.type`.

On one hand, `FileEntry` is more verbose now. On the other hand, discriminants give us several benefits – as we'll see soon.

19.1.3 Discriminated unions are related to algebraic data types

As an aside, discriminated unions are related to algebraic data types in functional programming languages. This is what `FileEntry` looks like as an algebraic data type in Haskell (if the TypeScript union elements had more properties, we'd probably use records in Haskell).

```
data FileEntry = FileEntryData String
  | FileEntryGenerator (String -> String)
  | FileEntryFile String
```

19.1.4 `readFile()` for the new `FileEnty`

Let's adapt `readFile()` to the new shape of `FileEnty`:

```
function readFile(vfs: VirtualFileSystem, path: string): string {
  const fileEntry = vfs.get(path);
  if (fileEntry === undefined) {
    throw new Error('Unknown path: ' + JSON.stringify(path));
  }
  switch (fileEntry.kind) {
    case 'FileEntryData':
      return fileEntry.data;
    case 'FileEntryGenerator':
      return fileEntry.generator(path);
    case 'FileEntryFile':
      return fs.readFileSync(fileEntry.path, 'utf-8');
    default:
```

```
        throw new UnexpectedValueError(fileEntry);
    }
  }
```

This brings us to a first advantage of discriminated unions: We can use `switch` statements. And it's immediately clear that `.kind` distinguishes the type union elements – we don't have to look for property names that are unique to elements.

Note that narrowing works as it did before: Once we have checked `.kind`, we can access all relevant properties.

19.1.5 Pros and cons of discriminated unions

- Con: Discriminating a union of object types makes it more verbose.
- Pro: We can handle cases via a `switch` statement.
- Pro: It's immediately clear which property distinguishes the elements of the union.

Pro: Inline union type elements come with descriptions

Another benefit is that, if the union elements are inlined (and not defined externally via types with names) then we can still see what each element does:

```
type Shape =
| {
  tag: 'Triangle',
  corner1: Point,
  corner2: Point,
  corner3: Point,
}
| {
  tag: 'Rectangle',
  corner1: Point,
  corner2: Point,
}
| {
  tag: 'Circle',
  center: Point,
  radius: number,
}
;
```

Pro: Union elements are not required to have unique properties

Discriminated unions work even if all normal properties of union elements are the same:

```
type Temperature =
  | {
    type: 'TemperatureCelsius',
    value: number,
  }
  | {
```

```
    type: 'TemperatureFahrenheit',
    value: number,
  }
;
```

General benefit of unions of object types: descriptiveness

The following type definition is terse; but can you tell how it works?

```
type OutputPathDef =
  | null // same as input path
  | '' // stem of output path
  | string // output path with different extension
```

If we use a discriminated union, the code becomes much more self-descriptive:

```
type OutputPathDef =
  | { key: 'sameAsInputPath' }
  | { key: 'inputPathStem' }
  | { key: 'inputPathStemPlusExt', ext: string }
  ;
```

This is a function that uses `OutputPathDef`:

```
import * as path from 'node:path';
function deriveOutputPath(def: OutputPathDef, inputPath: string): string {
  if (def.key === 'sameAsInputPath') {
    return inputPath;
  }
  const parsed = path.parse(inputPath);
  const stem = path.join(parsed.dir, parsed.name);
  switch (def.key) {
    case 'inputPathStem':
      return stem;
    case 'inputPathStemPlusExt':
      return stem + def.ext;
  }
}
const zip = { key: 'inputPathStemPlusExt', ext: '.zip' } as const;
assert.equal(
  deriveOutputPath(zip, '/tmp/my-dir'),
  '/tmp/my-dir.zip'
);
```

19.2 Deriving types from discriminated unions

In this section, we explore how we can derive types from discriminated unions. As an example, we work with the following discriminated union:

```
type Content =
  | {
```

```
    kind: 'text',
    charCount: number,
  }
  | {
    kind: 'image',
    width: number,
    height: number,
  }
  | {
    kind: 'video',
    width: number,
    height: number,
    runningTimeInSeconds: number,
  }
;
```

19.2.1 Extracting the values of the discriminant (the type tags)

To extract the values of the discriminant, we can use an indexed access type (`T[K]`):

```
type GetKind<T extends {kind: string}> =
  T['kind'];

type ContentKind = GetKind<Content>;

type _ = Assert<Equal<
  ContentKind,
  'text' | 'image' | 'video'
>>;
```

Because indexed access types are distributive over unions, `T['kind']` is applied to each element of `Content` and the result is a union of string literal types.

19.2.2 Maps for the elements of discriminated unions

If we use the type `ContentKind` from the previous subsection, we can define an exhaustive map for the elements of `Content`:

```
const DESCRIPTIONS_FULL: Record<ContentKind, string> = {
  text: 'plain text',
  image: 'an image',
  video: 'a video',
} as const;
```

If the map should not be exhaustive, we can use the utility type `Partial`:

```
const DESCRIPTIONS_PARTIAL: Partial<Record<ContentKind, string>> = {
  text: 'plain text',
} as const;
```

19.2.3 Extracting a subtype of a discriminated union

Sometimes, we don't need all of a discriminated union. We can write out own utility type for extracting a subtype of `Content`:

```
type ExtractSubtype<
  Union extends {kind: string},
  SubKinds extends GetKind<Union> // (A)
> =
  Union extends {kind: SubKinds} ? Union : never // (B)
;
```

We use a conditional type to loop over the union type `U`:

- Line B: If property `.kind` of a union element has a type that is assignable to `SubKinds` then we keep the element. If not then we omit it (by returning `never`).
- The `extends` in line A ensures that we don't make a typo when we extract: Our discriminant values `SubKinds` must be a subset of `GetKind<Union>` (see earlier subsection).

Let's use `ExtractSubtype`:

```
type _ = Assert<Equal<
  ExtractSubtype<Content, 'text' | 'image'>,
  | {
    kind: 'text',
    charCount: number,
  }
  | {
    kind: 'image',
    width: number,
    height: number,
  }
>>;
```

As an alternative to our own `ExtractSubtype`, we can also use the built-in utility type `Extract`:

```
type _ = Assert<Equal<
  Extract<Content, {kind: 'text' | 'image'}>,
  | {
    kind: 'text',
    charCount: number,
  }
  | {
    kind: 'image',
    width: number,
    height: number,
  }
>>;
```

`Extract` returns all elements of the union `Content` that are assignable to the following type:

```
{kind: 'text' | 'image'}
```

19.3 Class hierarchies vs. discriminated unions

To compare class hierarchies with discriminated unions, we use both to define syntax trees for representing expressions such as:

```
1 + 2 + 3
```

A syntax tree is either:

- A number value
- The addition of two syntax trees

19.3.1 A class hierarchy for syntax trees

The following code uses an abstract class and two subclasses to represent syntax trees:

```
abstract class SyntaxTree {
  abstract evaluate(): number;
}

class NumberValue extends SyntaxTree {
  numberValue: number;
  constructor(numberValue: number) {
    super();
    this.numberValue = numberValue;
  }
  evaluate(): number {
    return this.numberValue;
  }
}
class Addition extends SyntaxTree {
  operand1: SyntaxTree;
  operand2: SyntaxTree;
  constructor(operand1: SyntaxTree, operand2: SyntaxTree) {
    super();
    this.operand1 = operand1;
    this.operand2 = operand2;
  }
  evaluate(): number {
    return this.operand1.evaluate() + this.operand2.evaluate();
  }
}
```

The operation `evaluate` handles the two cases "number value" and "addition" in the corresponding classes – via polymorphism. Here it is in action:

```
const syntaxTree = new Addition(
  new NumberValue(1),
  new Addition(
```

```
    new NumberValue(2),
    new NumberValue(3),
  ),
);
assert.equal(
  syntaxTree.evaluate(), 6
);
```

19.3.2 A discriminated union for syntax trees

The following code uses a discriminated union with two elements to represent syntax trees:

```
type SyntaxTree =
  | {
    kind: 'NumberValue';
    numberValue: number;
  }
  | {
    kind: 'Addition';
    operand1: SyntaxTree;
    operand2: SyntaxTree;
  }
;

function evaluate(syntaxTree: SyntaxTree): number {
  switch(syntaxTree.kind) {
    case 'NumberValue':
      return syntaxTree.numberValue;
    case 'Addition':
      return (
        evaluate(syntaxTree.operand1) +
        evaluate(syntaxTree.operand2)
      );
    default:
      throw new UnexpectedValueError(syntaxTree);
  }
}
```

The operation `evaluate` handles the two cases "number value" and "addition" in a single location, via `switch`. Here it is in action:

```
const syntaxTree: SyntaxTree = {
  kind: 'Addition',
  operand1: {
    kind: 'NumberValue',
    numberValue: 1,
  },
  operand2: {
    kind: 'Addition',
    operand1: {
```

```
      kind: 'NumberValue',
      numberValue: 2,
    },
    operand2: {
      kind: 'NumberValue',
      numberValue: 3,
    },
  }
};
assert.equal(
  evaluate(syntaxTree), 6
);
```

We don't need the type annotation in line A, but it helps ensure that the data has the correct structure. If we don't do it here, we'll find out about problems later.

19.3.3 Comparing classes and discriminated unions

With classes, we check the types of instances via `instanceof`. With discriminated unions, we use discriminants to do so. In a way, they are runtime type information.

Each approach does one kind of extensibility well:

- With classes, we have to modify each class if we want to add a new operation. However, adding a new type does not require any changes to existing code.
- With discriminated unions, we have to modify each function if we want to add a new type. In contrast, adding new operations is simple.

19.4 Defining discriminated unions via classes

It's also possible to define a discriminated union via classes – e.g.:

```
type Color = Black | White;

abstract class AbstractColor {}
class Black extends AbstractColor {
  readonly kind = 'Black';
}
class White extends AbstractColor {
  readonly kind = 'White';
}

function colorToRgb(color: Color): string {
  switch (color.kind) {
    case 'Black':
      return '#000000';
    case 'White':
      return '#FFFFFF';
```

```
    }
  }
```

Why would we want to do that? We can define and inherit methods for the elements of the union.

The abstract class `AbstractColor` is only needed if we want to share methods between the union classes.

Chapter 20

Intersections of object types

20.1 Intersections of object types 187
20.1.1 Extending vs. intersection 187
20.1.2 Example: `NonNullable` (`T & {}`) 188
20.1.3 Example: inferred intersections 189
20.1.4 Example: combining two object types via an intersection 189

In this chapter, we explore what intersections of object types can be used for in TypeScript.

In this chapter, *object type* means:

- Object literal type
- Interface type
- Mapped type (such as `Record`)

20.1 Intersections of object types

The intersection of two object types has the properties of both:

```
type Obj1 = { prop1: boolean };
type Obj2 = { prop2: number };

const obj: Obj1 & Obj2 = {
  prop1: true,
  prop2: 123,
};
```

20.1.1 Extending vs. intersection

With interfaces, we can use `extends` to add properties:

```
interface Person {
  name: string;
}
interface Employe extends Person {
  company: string;
}
```

With object types, we can use an intersection:

```
type Person = {
  name: string,
};

type Employee =
  & Person
  & {
    company: string,
  }
;
```

One caveat is that only `extends` supports overriding. For more information, see $type.

20.1.2 Example: `NonNullable` (`T & {}`)

```
/**
 * Exclude null and undefined from T
 */
type NonNullable<T> = T & {};

type _ = [
  Assert<Equal<
    NonNullable<undefined | string>,
    string
  >>,
  Assert<Equal<
    NonNullable<null | string>,
    string
  >>,
  Assert<Equal<
    NonNullable<string>, // (A)
    string
  >>,
];
```

The result of `NonNullable<T>` is a type that is the intersection of `T` and all non-nullish values. It's interesting that `string & {}` is `string` (line A).

20.1.3 Example: inferred intersections

The following code shows how the inferred type of `obj` changes when we use the built-in type guard `in` (line A and line B):

```
function func(obj: object) {
  if ('prop1' in obj) { // (A)
    assertType<
      object & Record<'prop1', unknown>
    >(obj);
    if ('prop2' in obj) { // (B)
      assertType<
      object & Record<'prop1', unknown> & Record<'prop2', unknown>
      >(obj);
    }
  }
}
```

20.1.4 Example: combining two object types via an intersection

In the next example, we combine the type `Obj` of a parameter with the type `WithKey` – by adding the property `.key` of `WithKey` to the parameter:

```
type WithKey = {
  key: string,
};
function addKey<Obj extends object>(obj: Obj, key: string)
  : Obj & WithKey
{
  const objWithKey = obj as (Obj & WithKey);
  objWithKey.key = key;
  return objWithKey;
}
```

`addKey()` is used like this:

```
const paris = {
  city: 'Paris',
};

const parisWithKey = addKey(paris, 'paris');
assertType<
{
  city: string,
  key: string,
}
>(parisWithKey);
```

Chapter 21

Class definitions in TypeScript

21.1 Cheat sheet: classes in plain JavaScript 192
21.1.1 Basic members of classes 192
21.1.2 Modifier: `static` 192
21.1.3 Modifier-like name prefix: `#` (private) 192
21.1.4 Modifiers for accessors: `get` (getter) and `set` (setter) 193
21.1.5 Modifier for methods: `*` (generator) 193
21.1.6 Modifier for methods: `async` 194
21.1.7 Computed class member names 194
21.1.8 Combinations of modifiers 194
21.1.9 Under the hood 195
21.1.10 More information on class definitions in plain JavaScript 196
21.2 Non-public data slots in TypeScript 196
21.2.1 Private properties 197
21.2.2 Private fields 197
21.2.3 Private properties vs. private fields 198
21.2.4 Protected properties 199
21.3 Private constructors 200
21.4 Initializing instance properties 201
21.4.1 Strict property initialization 201
21.5 Convenience features we should avoid 202
21.5.1 Inferred member types 202
21.5.2 Making constructor parameters `public`, `private` or `protected` 202
21.6 Abstract classes 203
21.7 Keyword `override` for methods 205
21.8 Classes vs. object types 205
21.8.1 Class `Counter` 205
21.8.2 Object type `Counter` 206
21.8.3 Which one to choose: class or object type? 206

In this chapter, we examine how class definitions work in TypeScript:

- First, we take a quick look at the features of class definitions in plain JavaScript.
- Then we explore what additions TypeScript brings to the table.

21.1 Cheat sheet: classes in plain JavaScript

This section is a cheat sheet for class definitions in plain JavaScript.

21.1.1 Basic members of classes

```
class OtherClass {}

class MyClass1 extends OtherClass {
  publicInstanceField = 1;
  constructor() {
    super();
  }
  publicPrototypeMethod() {
    return 2;
  }
}

const inst1 = new MyClass1();
assert.equal(inst1.publicInstanceField, 1);
assert.equal(inst1.publicPrototypeMethod(), 2);
```

The next sections are about modifiers

At the end, there is a table that shows how modifiers can be combined.

21.1.2 Modifier: `static`

```
class MyClass2 {
  static staticPublicField = 1;
  static staticPublicMethod() {
    return 2;
  }
}

assert.equal(MyClass2.staticPublicField, 1);
assert.equal(MyClass2.staticPublicMethod(), 2);
```

21.1.3 Modifier-like name prefix: # (private)

```
class MyClass3 {
  #privateField = 1;
```

```
  #privateMethod() {
    return 2;
  }
  static accessPrivateMembers() {
    // Private members can only be accessed from inside class definitions
    const inst3 = new MyClass3();
    assert.equal(inst3.#privateField, 1);
    assert.equal(inst3.#privateMethod(), 2);
  }
}
```

21.1.4 Modifiers for accessors: `get` (getter) and `set` (setter)

Roughly, accessors are prototype methods that are inherited by instances and invoked by accessing properties. There are two kinds of accessors: getters and setters.

```
class MyClass4 {
  #name = 'Rumpelstiltskin';

  /** Prototype getter */
  get name() {
    return this.#name;
  }

  /** Prototype setter */
  set name(value) {
    this.#name = value;
  }
}
const inst5 = new MyClass4();
assert.equal(inst5.name, 'Rumpelstiltskin'); // getter
inst5.name = 'Queen'; // setter
assert.equal(inst5.name, 'Queen'); // getter
```

21.1.5 Modifier for methods: `*` (generator)

```
class MyClass5 {
  * publicPrototypeGeneratorMethod() {
    yield 'hello';
    yield 'world';
  }
}

const inst6 = new MyClass5();
assert.deepEqual(
  Array.from(inst6.publicPrototypeGeneratorMethod()),
  ['hello', 'world']
);
```

21.1.6 Modifier for methods: async

```
class MyClass6 {
  async publicPrototypeAsyncMethod() {
    const result = await Promise.resolve('abc');
    return result + result;
  }
}

const inst7 = new MyClass6();
assert.equal(
  await inst7.publicPrototypeAsyncMethod(),
  'abcabc'
);
```

21.1.7 Computed class member names

```
const publicInstanceFieldKey = Symbol('publicInstanceFieldKey');
const publicPrototypeMethodKey = Symbol('publicPrototypeMethodKey');

class MyClass7 {
  [publicInstanceFieldKey] = 1;
  [publicPrototypeMethodKey]() {
    return 2;
  }
}

const inst8 = new MyClass7();
assert.equal(inst8[publicInstanceFieldKey], 1);
assert.equal(inst8[publicPrototypeMethodKey](), 2);
```

Comments:

- The main use case for this feature is symbols such as `Symbol.iterator`. But any expression can be used inside the square brackets.
- We can compute the names of fields, methods, and accessors.
- We cannot compute the names of private members (which are always fixed).

21.1.8 Combinations of modifiers

Fields:

Level	Private	Code
(instance)		`field`
(instance)	#	`#field`
`static`		`static field`
`static`	#	`static #field`

Methods (columns: Level, Accessor, Async, Generator, Private, Code – without body):

Level	Acc	Async	Gen	Priv	Code
(prototype)					`m()`
(prototype)	`get`				`get p()`
(prototype)	`set`				`set p(x)`
(prototype)		`async`			`async m()`
(prototype)			`*`		`* m()`
(prototype)		`async`	`*`		`async * m()`
(prototype-ish)				`#`	`#m()`
(prototype-ish)	`get`			`#`	`get #p()`
(prototype-ish)	`set`			`#`	`set #p(x)`
(prototype-ish)		`async`		`#`	`async #m()`
(prototype-ish)			`*`	`#`	`* #m()`
(prototype-ish)		`async`	`*`	`#`	`async * #m()`
`static`					`static m()`
`static`	`get`				`static get p()`
`static`	`set`				`static set p(x)`
`static`		`async`			`static async m()`
`static`			`*`		`static * m()`
`static`		`async`	`*`		`static async * m()`
`static`				`#`	`static #m()`
`static`	`get`			`#`	`static get #p()`
`static`	`set`			`#`	`static set #p(x)`
`static`		`async`		`#`	`static async #m()`
`static`			`*`	`#`	`static * #m()`
`static`		`async`	`*`	`#`	`static async * #m()`

21.1.9 Under the hood

It's important to keep in mind that with classes, there are two chains of prototype objects:

- The instance chain which starts with an instance.
- The static chain which starts with the class of that instance.

Consider the following plain JavaScript example:

```
class ClassA {
  static staticMthdA() {}
  constructor(instPropA) {
    this.instPropA = instPropA;
  }
  prototypeMthdA() {}
}
class ClassB extends ClassA {
  static staticMthdB() {}
  constructor(instPropA, instPropB) {
    super(instPropA);
    this.instPropB = instPropB;
  }
```

```
  prototypeMthdB() {}
}
const instB = new ClassB(0, 1);
```

Figure 21.1 shows what the prototype chains look like that are created by `ClassA` and `ClassB`.

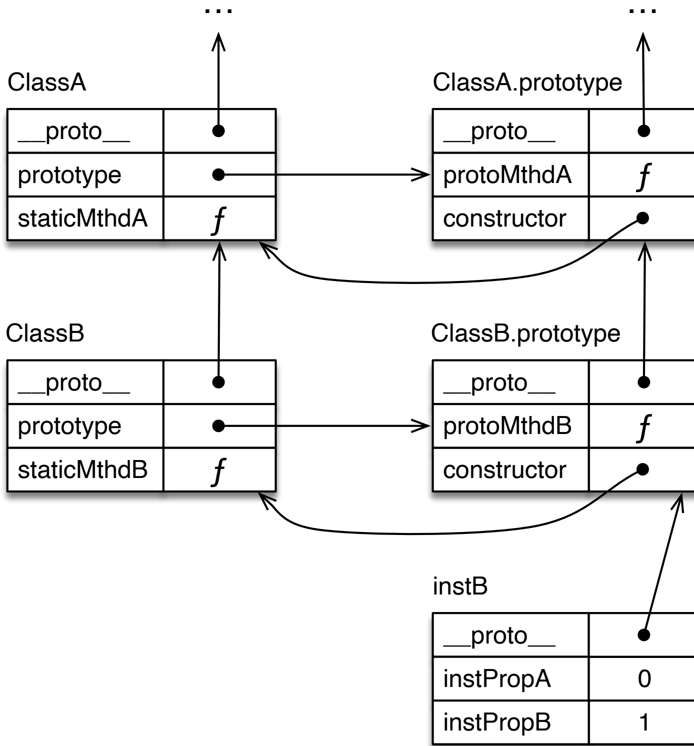

Figure 21.1: The classes `ClassA` and `ClassB` create two prototype chains: One for classes (left-hand side) and one for instances (right-hand side).

21.1.10 More information on class definitions in plain JavaScript

- Chapter "Classes" in "Exploring JavaScript"

21.2 Non-public data slots in TypeScript

By default, all data slots in TypeScript are public properties. There are two ways of keeping data private:

- Private properties
- Private fields

We'll look at both next.

Note that TypeScript does not currently support private methods.

21.2.1 Private properties

Private properties are a TypeScript-only (static) feature. Any property can be made private by prefixing it with the keyword `private` (line A):

```
class PersonPrivateProperty {
  private name: string; // (A)
  constructor(name: string) {
    this.name = name;
  }
  sayHello() {
    return `Hello ${this.name}!`;
  }
}
```

We now get compile-time errors if we access that property in the wrong scope (line A):

```
const john = new PersonPrivateProperty('John');

assert.equal(
  john.sayHello(), 'Hello John!'
);

// @ts-expect-error: Property 'name' is private and only accessible
// within class 'PersonPrivateProperty'.
john.name; // (A)
```

However, `private` doesn't change anything at runtime. There, property `.name` is indistinguishable from a public property:

```
assert.deepEqual(
  Object.keys(john),
  ['name']
);
```

We can also see that private properties aren't protected at runtime when we look at the JavaScript code that the class is compiled to:

```
class PersonPrivateProperty {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    return `Hello ${this.name}!`;
  }
}
```

21.2.2 Private fields

Private fields are a new JavaScript feature that TypeScript has supported since version 3.8:

```
class PersonPrivateField {
  #name: string;
  constructor(name: string) {
    this.#name = name;
  }
  sayHello() {
    return `Hello ${this.#name}!`;
  }
}
```

This version of `Person` is mostly used the same way as the private property version:

```
const john = new PersonPrivateField('John');

assert.equal(
  john.sayHello(), 'Hello John!'
);
```

However, this time, the data is completely encapsulated. Using the private field syntax outside classes is even a JavaScript syntax error. That's why we have to use `eval()` in line A so that we can execute this code:

```
assert.throws(
  () => eval('john.#name'), // (A)
  {
    name: 'SyntaxError',
    message: "Private field '#name' must be declared in "
      + "an enclosing class",
  }
);

assert.deepEqual(
  Object.keys(john),
  []
);
```

Compiled to JavaScript, `PersonPrivateField` looks more or less the same:

```
class PersonPrivateField {
  #name;
  constructor(name) {
    this.#name = name;
  }
  sayHello() {
    return `Hello ${this.#name}!`;
  }
}
```

21.2.3 Private properties vs. private fields

- Downsides of private properties:

    - We can't reuse the names of private properties in subclasses (because the properties aren't private at runtime).
    - No encapsulation at runtime.
- Upsides of private properties:
    - Clients can circumvent the encapsulation and access private properties. This can be useful if someone needs to work around a bug. In other words: Data being completely encapsulated has pros and cons.
    - Some JavaScript helper functions, e.g. for cloning or for serialization to JSON, don't work with private fields.

21.2.4 Protected properties

Private fields and private properties can't be accessed in subclasses (line B):

```
class PrivatePerson {
  private name: string;
  constructor(name: string) {
    this.name = name;
  }
  sayHello() {
    return `Hello ${this.name}!`;
  }
}
class PrivateEmployee extends PrivatePerson {
  private company: string;
  constructor(name: string, company: string) {
    super(name);
    this.company = company;
  }
  override sayHello() { // (A)
    // @ts-expect-error: Property 'name' is private and only
    // accessible within class 'PrivatePerson'.
    return `Hello ${this.name} from ${this.company}!`; // (B)
  }
}
```

The keyword `override` is explained later – it's for methods that override super-methods.

We can fix the previous example by switching from `private` to `protected` in line A (we also switch in line B, for consistency's sake):

```
class ProtectedPerson {
  protected name: string; // (A)
  constructor(name: string) {
    this.name = name;
  }
  sayHello() {
    return `Hello ${this.name}!`;
  }
}
```

```
class ProtectedEmployee extends ProtectedPerson {
  protected company: string; // (B)
  constructor(name: string, company: string) {
    super(name);
    this.company = company;
  }
  override sayHello() {
    return `Hello ${this.name} from ${this.company}!`; // OK
  }
}
```

21.3 Private constructors

At the moment, JavaScript does not support hash-private constructors. However, TypeScript supports `private` for them. That is useful when we have static factory methods and want clients to always use those methods, never the constructor directly. Static methods can access private class members, which is why the factory methods can still use the constructor.

In the following code, there is one static factory method `DataContainer.create()`. It sets up instances via asynchronously loaded data. Keeping the asynchronous code in the factory method enables the actual class to be completely synchronous:

```
class DataContainer {
  #data: string;
  static async create() {
    const data = await Promise.resolve('downloaded'); // (A)
    return new this(data);
  }
  private constructor(data: string) {
    this.#data = data;
  }
  getData() {
    return 'DATA: '+this.#data;
  }
}
const dataContainer = await DataContainer.create();
assert.equal(
  dataContainer.getData(),
  'DATA: downloaded'
);
```

In real-world code, we would use `fetch()` or a similar Promise-based API to load data asynchronously in line A.

The private constructor prevents `DataContainer` from being subclassed. If we want to allow subclasses, we have to make it `protected`.

21.4 Initializing instance properties

21.4.1 Strict property initialization

If the compiler setting `--strictPropertyInitialization` is switched on (which is the case if we use `--strict`), then TypeScript checks if all declared instance properties are correctly initialized:

- Either via assignments in the constructor:

  ```
  class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
      this.x = x;
      this.y = y;
    }
  }
  ```

- Or via initializers for the property declarations:

  ```
  class Point {
    x = 0;
    y = 0;

    // No constructor needed
  }
  ```

However, sometimes we initialize properties in a manner that TypeScript doesn't recognize. Then we can use exclamation marks (*definite assignment assertions*) to switch off TypeScript's warnings (line A and line B):

```
class Point {
  x!: number; // (A)
  y!: number; // (B)
  constructor() {
    this.initProperties();
  }
  initProperties() {
    this.x = 0;
    this.y = 0;
  }
}
```

Example: setting up instance properties via objects

In the following example, we also need definite assignment assertions. Here, we set up instance properties via the constructor parameter `props`:

```
class CompilerError implements CompilerErrorProps { // (A)
  line!: number;
  description!: string;
```

```
  constructor(props: CompilerErrorProps) {
    Object.assign(this, props); // (B)
  }
}

// Helper interface for the parameter properties
interface CompilerErrorProps {
  line: number,
  description: string,
}

// Using the class:
const err = new CompilerError({
  line: 123,
  description: 'Unexpected token',
});
```

Notes:

- In line B, we initialize all properties: We use `Object.assign()` to copy the properties of parameter `props` into `this`.
- In line A, the `implements` ensures that the class declares all properties that are part of interface `CompilerErrorProps`.

21.5 Convenience features we should avoid

21.5.1 Inferred member types

`tsc` can infer the type of the member `.str` because we assign to it in line A. However, that is not compatible with the compiler option `isolatedDeclarations` (which enables external tools to generate declarations without doing inference):

```
class C {
  str;
  constructor(str: string) {
    this.str = str; // (A)
  }
}
```

21.5.2 Making constructor parameters `public`, `private` or `protected`

JavaScript currently has no equivalent to the TypeScript feature described in this subsection – which is why it is illegal if the compiler option `erasableSyntaxOnly` is active.

If we use the modifier `public` for a constructor parameter `prop`, then TypeScript does two things for us:

- It declares a public instance property `.prop`.
- It assigns the parameter `prop` to that instance property.

This is an example:

```
class Point {
  constructor(public x: number, public y: number) {
  }
}
```

If we use `private` or `protected` instead of `public`, then the corresponding instance properties are private or protected.

The TypeScript class `Point` is compiled to the following JavaScript code:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
```

21.6 Abstract classes

Two constructs can be abstract in TypeScript:

- An abstract class can't be instantiated. Only its subclasses can – if they are not abstract, themselves.
- An abstract method has no implementation, only a type signature. Each concrete subclass must have a concrete method with the same name and a compatible type signature.
  - If a class has any abstract methods, it must be abstract, too.

The following code demonstrates abstract classes and methods.

On one hand, there is the abstract superclass `Printable` and its helper class `StringBuilder`:

```
class StringBuilder {
  string = '';
  add(str: string) {
    this.string += str;
  }
}
abstract class Printable {
  toString() {
    const out = new StringBuilder();
    this.print(out);
    return out.string;
  }
  abstract print(out: StringBuilder): void;
}
```

On the other hand, there are the concrete subclasses `Entries` and `Entry`:

```
class Entries extends Printable {
  entries: Entry[];
  constructor(entries: Entry[]) {
```

```
    super();
    this.entries = entries;
  }
  print(out: StringBuilder): void {
    for (const entry of this.entries) {
      entry.print(out);
    }
  }
}
class Entry extends Printable {
  key: string;
  value: string;
  constructor(key: string, value: string) {
    super();
    this.key = key;
    this.value = value;
  }
  print(out: StringBuilder): void {
    out.add(this.key);
    out.add(': ');
    out.add(this.value);
    out.add('\n');
  }
}
```

And finally, this is us using `Entries` and `Entry`:

```
const entries = new Entries([
  new Entry('accept-ranges', 'bytes'),
  new Entry('content-length', '6518'),
]);
assert.equal(
  entries.toString(),
  'accept-ranges: bytes\ncontent-length: 6518\n'
);
```

Notes about abstract classes:

- An abstract class can be seen as an interface where some members already have implementations.
- While a class can implement multiple interfaces, it can only extend at most one abstract class.
- "Abstractness" only exists at compile time. At runtime, abstract classes are normal classes and abstract methods don't exist (due to them only providing compile-time information).
- Abstract classes can be seen as templates where each abstract method is a blank that has to be filled in (implemented) by subclasses.

21.7 Keyword `override` for methods

The keyword `override` is for methods that override methods in superclasses – e.g.:

```
class A {
  m(): void {}
}
class B extends A {
  // `override` is required
  override m(): void {} // (A)
}
```

If the compiler option `noImplicitOverride` is active then TypeScript complains if there is no `override` in line A.

We can also use `override` when we implement an abstract method. That's not required but I find it useful information:

```
abstract class A {
  abstract m(): void;
}
class B extends A {
  // `override` is optional
  override m(): void {}
}
```

21.8 Classes vs. object types

In JavaScript, we don't have to use classes, we can also use objects directly. TypeScript supports both approaches.

21.8.1 Class `Counter`

This is a class that implements a counter:

```
class Counter {
  count = 0;
  inc(): void {
    this.count++;
  }
}

// Trying out the functionality
const counter = new Counter();
counter.inc();
assert.equal(
  counter.count, 1
);
```

21.8.2 Object type `Counter`

In TypeScript, a class defines both a type and a factory for instances. In the following code, both are separate: We have the object type `Counter` and the factory `createCounter()`.

```
type Counter = {
  count: number,
};
function createCounter(): Counter {
  return {
    count: 0,
  };
}
function inc(counter: Counter): void {
  counter.count++;
}

// Trying out the functionality
const counter = createCounter();
inc(counter);
assert.equal(
  counter.count, 1
);
```

21.8.3 Which one to choose: class or object type?

Benefits of classes:

- Everything is specified compactly in one place:
  - Type
  - Instance factory
  - Operations such as `inc`
  - The default value of a property being specified close to the definition of that property is something I find useful – e.g., `.count` has the default value 0.
- We can check the type of a value via `instanceof` – e.g. to narrow a type.
- We can use private fields.

Benefits of object types:

- They work better if objects are cloned: Library functions for cloning can't handle private fields and `structuredClone()` does not preserve the class of an instance.
- They work better if objects are moved between realms: Each realm has its own version of a given class and that makes moving class instances problematic.

Serializing and deserializing (to/from JSON etc.) is an interesting use case:

- With object types, deserialization is easier because we can immediate work with the result of `JSON.parse()` (potentially after validating the type via Zod).
- Things get more complicated if not all data can be easily serialized and deserialized – e.g. if a property contains a `Map`. Then classes have one benefit: We can customize serialization by implementing the method `.toJSON()`.

Apart from these criteria, which one to choose depends on whether you prefer code that is more object-oriented or code that is more functional.

We have not covered inheritance – where you also have a choice between an object-oriented coding style (classes) and a functional coding style (discriminated unions). For more information, see “Class hierarchies vs. discriminated unions” (§19.3).

Chapter 22

Class-related types

22.1 The two prototype chains of classes 209
22.2 Interfaces for instances of classes 210
22.3 Interfaces for classes 211
22.3.1 Example: converting from and to JSON 211
22.3.2 Example: TypeScript's built-in interfaces for the class `Object` and for its instances 213
22.4 Classes as types 213
22.4.1 Pitfall: classes work structurally, not nominally 214
22.5 Further reading 216

In this chapter, we examine types related to classes and their instances.

22.1 The two prototype chains of classes

Consider this class:

```
class Counter extends Object {
  static createZero() {
    return new Counter(0);
  }
  value: number;
  constructor(value: number) {
    super();
    this.value = value;
  }
  increment() {
    this.value++;
  }
}
```

```
// Static method
const myCounter = Counter.createZero();
assert.ok(myCounter instanceof Counter);
assert.equal(myCounter.value, 0);

// Instance method
myCounter.increment();
assert.equal(myCounter.value, 1);
```

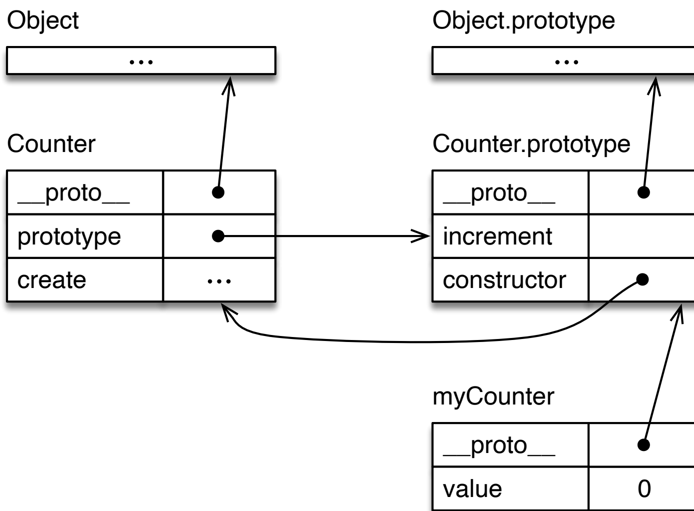

Figure 22.1: Objects created by class `Counter`. Left-hand side: the class and its superclass `Object`. Right-hand side: The instance `myCounter`, the prototype properties of `Counter`, and the prototype methods of the superclass `Object`.

The diagram in figure 22.1 shows the runtime structure of class `Counter`. There are two prototype chains of objects in this diagram:

- Class (left-hand side): The static prototype chain consists of the objects that make up class `Counter`. The prototype object of class `Counter` is its superclass, `Object`.
- Instance (right-hand side): The instance prototype chain consists of the objects that make up the instance `myCounter`. The chain starts with the instance `myCounter` and continues with `Counter.prototype` (which holds the prototype methods of class `Counter`) and `Object.prototype` (which holds the prototype methods of class `Object`).

In this chapter, we'll first explore instance objects and then classes as objects.

22.2 Interfaces for instances of classes

Interfaces specify services that objects provide. For example:

```
interface CountingService {
  value: number;
```

```
  increment(): void;
}
```

TypeScript's interfaces work structurally: In order for an object to implement an interface, it only needs to have the right properties with the right types. We can see that in the following example:

```
const myCounter2: CountingService = new Counter(3);
```

Structural interfaces are convenient because we can create interfaces even for objects that already exist (i.e., we can introduce them after the fact).

If we know ahead of time that an object must implement a given interface, it often makes sense to check early if it does, in order to avoid surprises later. We can do that for instances of classes via `implements`:

```
class Counter implements CountingService {
  // ···
};
```

Comments:

- We can `implement` any object type (not just interfaces).
- TypeScript does not distinguish between inherited properties (such as `.increment`) and own properties (such as `.value`).
- As an aside, private properties are ignored by interfaces and can't be specified via them. This is expected given that private data is for internal purposes only.

22.3 Interfaces for classes

Classes themselves are also objects (functions). Therefore, we can use interfaces to specify their properties. The main use case here is describing factories for objects. The next section gives an example.

22.3.1 Example: converting from and to JSON

The following two interfaces can be used for classes that support their instances being converted from and to JSON:

```
// Converting JSON to instances
interface JsonStatic {
  fromJson(json: unknown): JsonInstance;
}

// Converting instances to JSON
interface JsonInstance {
  toJson(): unknown;
}
```

We use these interfaces in the following code:

```
class Person implements JsonInstance {
  static fromJson(json: unknown): Person {
    if (typeof json !== 'string') {
      throw new TypeError();
    }
    return new Person(json);
  }
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  toJson(): unknown {
    return this.name;
  }
}
```

This is how we can check right away if class `Person` (as an object) implements the interface `JsonStatic`:

```
type _ = Assert<Assignable<JsonStatic, typeof Person>>;
```

If you don't want to use a library (with the utility types `Assert` and `Assignable`) for this purpose, you can use the following pattern:

```
// Assign the class to a type-annotated variable
const personImplementsJsonStatic: JsonStatic = Person;
```

The downside of this pattern is that it produces extra JavaScript code.

Can we do better?

It would be nice to avoid an external check – e.g., like this:

```
const Person = class implements JsonInstance {
  static fromJson(json: unknown): Person { // (A)
    // ···
  }
  // ···
} satisfies JsonStatic; // (B)
type Person = typeof Person.prototype; // (C)
```

In line B, we use the `satisfies` operator, which enforces that the value `Person` is assignable to `JsonStatic` while preserving the type of that value. That is important because `Person` should not be limited to what's defined in `JsonStatic`.

Alas, this alternative approach is even more verbose and doesn't compile. One of the compiler errors is in line C:

> Type alias 'Person' circularly references itself.

Why? Type `Person` is mentioned in line A. Even if we rename the type `Person` to `TPerson`, that error doesn't go away.

22.3.2 Example: TypeScript's built-in interfaces for the class `Object` and for its instances

It is instructive to take a look at TypeScript's built-in types:

On one hand, interface `ObjectConstructor` is for the class pointed to by the global variable `Object`:

```
declare var Object: ObjectConstructor;
interface ObjectConstructor {
  /** Invocation via `new` */
  new(value?: any): Object; // (A)
  /** Invocation via function calls */
  (value?: any): any;

  readonly prototype: Object; // (B)

  getPrototypeOf(o: any): any;
  // ···
}
```

On the other hand, interface `Object` (which is mentioned in line A and line B) is for instances of `Object`:

```
interface Object {
  constructor: Function;
  toString(): string;
  toLocaleString(): string;
  valueOf(): Object;
  hasOwnProperty(v: PropertyKey): boolean;
  isPrototypeOf(v: Object): boolean;
  propertyIsEnumerable(v: PropertyKey): boolean;
}
```

In other words – the name `Object` is used twice, at two different language levels:

- At the dynamic level, for a global variable.
- At the static level, for a type.

22.4 Classes as types

Consider the following class:

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

This class definition creates two things.

First, a constructor function named `Color` (that can be invoked via `new`):

```
assert.equal(
  typeof Color, 'function'
);
```

Second, an interface named `Color` that matches instances of `Color`:

```
const green: Color = new Color('green');
```

Here is proof that `Color` really is an interface:

```
interface RgbColor extends Color {
  rgbValue: [number, number, number];
}
```

22.4.1 Pitfall: classes work structurally, not nominally

There is one pitfall, though: Using `Color` as a static type is not a very strict check:

```
class Color {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}

const person: Person = new Person('Jane');
const color: Color = person; // (A)
```

Why doesn't TypeScript complain in line A? That's due to structural typing: Instances of `Person` and of `Color` have the same structure and are therefore statically compatible.

Switching off structural typing

We can turn `Color` into a nominal type by adding a private field (or a `private` property):

```
class Color {
  name: string;
  #isBranded = true;
  constructor(name: string) {
    this.name = name;
  }
}
class Person {
  name: string;
```

```
  #isBranded = true;
  constructor(name: string) {
    this.name = name;
  }
}

const robin: Person = new Person('Robin');
// @ts-expect-error: Type 'Person' is not assignable to type 'Color'.
// Property '#isBranded' in type 'Person' refers to a different member that
// cannot be accessed from within type 'Color'.
const color: Color = robin;
```

This way of switching off structural typing is called *branding*. Note that the private fields of `Color` and `Person` are incompatible even though they have the same name and the same type. That reflects how JavaScript works: We cannot access the private field of `Color` from `Person` and vice versa.

Use case for branding: migrating from an object type to a class

Let's say we want to migrate the following code from the object type in line A to a class:

```
type Person = { // (A)
  name: string,
};

function storePerson(person: Person): void {
  // ...
}

storePerson({
  name: 'Robin',
});
```

In our first attempt, invoking `storePerson()` with an object literal still works:

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

function storePerson(person: Person): void {
  // ...
}

storePerson({
  name: 'Robin',
});
```

Once we brand `Person`, we get a compiler error:

```
class Person {
  name: string;
  #isBranded = true;
  constructor(name: string) {
    this.name = name;
  }
}

function storePerson(person: Person): void {
  // ...
}

// @ts-expect-error: Argument of type '{ name: string; }' is not assignable
// to parameter of type 'Person'. Property '#isBranded' is missing in type
// '{ name: string; }' but required in type 'Person'.
storePerson({
  name: 'Robin',
});
```

This is how we fix the error:

```
storePerson(
  new Person('Robin')
);
```

22.5 Further reading

- Chapter "Classes" in "Exploring JavaScript"

Chapter 23

Types for classes as values

23.1 Question: Which type for a class as a value? . . . 217
23.2 Answer: types for classes as values . . . 218
  23.2.1 The type operator `typeof` . . . 218
  23.2.2 Constructor type literals . . . 218
  23.2.3 Object type literals with construct signatures . . . 219
23.3 A generic type for constructors: `Class<T>` . . . 219
  23.3.1 Example: creating instances . . . 219
  23.3.2 Example: type-narrowing via `instanceof` . . . 220
  23.3.3 Example: casting with runtime checks . . . 220
  23.3.4 Example: an assertion function . . . 221
  23.3.5 Example: Maps that are type-safe at runtime . . . 221
  23.3.6 Pitfall: `Class<T>` does not match abstract classes . . . 222

In this chapter, we explore classes as values:

- What types should we use for such values?
- What are the use cases for these types?

23.1 Question: Which type for a class as a value?

Consider the following class:

```
class Point {
  x: number;
  y: number;
  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
```

```
  }
}
```

This function accepts a class and creates an instance of it:

```
function createPoint(PointClass: C, x: number, y: number): Point {
  return new PointClass(x, y);
}
```

What type `C` should we use for the parameter `PointClass` if we want the function to return an instance of `Point`?

23.2 Answer: types for classes as values

23.2.1 The type operator `typeof`

In "TypeScript's two language levels" (§4.4), we explored the two language levels of TypeScript:

- Dynamic level: JavaScript (code and values)
- Static level: TypeScript (static types)

The class `Point` creates two things:

- The constructor function `Point`
- The interface `Point` for instances of `Point`

Depending on where we mention `Point`, it means different things. That's why we can't use the type `Point` for `PointClass`: It matches *instances* of class `Point`, not class `Point` itself.

Instead, we need to use the type operator `typeof` (which has the same name as a JavaScript operator). `typeof v` stands for the type of the value `v`.

Let's omit the return type of `createPoint()` and see what TypeScript infers:

```
function createPoint(PointClass: typeof Point, x: number, y: number) {
  return new PointClass(x, y);
}

const point = createPoint(Point, 3, 6);
assertType<Point>(point); // (A)
assert.ok(point instanceof Point);
```

As expected, `createPoint()` creates values of type `Point` (line A).

23.2.2 Constructor type literals

A *constructor type literal* is a literal for constructor types: `new` followed by a function type literal (line A):

```
function createPoint(
  PointClass: new (x: number, y: number) => Point, // (A)
  x: number, y: number
```

```
) {
  return new PointClass(x, y);
}
```

The prefix `new` of its type indicates that `PointClass` is a function that must be invoked via `new`.

Constructor type literals are quite versatile – e.g., we can demand that a constructor function (such as a class):

- Have particular parameters.
- Return instances with a particular interface (see code below)

```
function f(
  ClassThatImplementsInterf: new () => Interf
) {}
```

23.2.3 Object type literals with construct signatures

Recall that members of interfaces and object literal types (OLTs) include method signatures and call signatures. Call signatures enable interfaces and OLTs to describe functions.

Similarly, *construct signatures* enable interfaces and OLTs to describe constructor functions. They look like call signatures with the added prefix `new`. In the next example, `PointClass` has an object literal type with a construct signature:

```
function createPoint(
  PointClass: {new (x: number, y: number): Point},
  x: number, y: number
) {
  return new PointClass(x, y);
}
```

23.3 A generic type for constructors: `Class<T>`

With the knowledge we have acquired, we can now create a generic type for classes as values – by introducing a type parameter `T`:

```
type Class<T> = new (...args: any[]) => T;
```

Instead of a type alias, we can also use an interface:

```
interface Class<T> {
  new(...args: any[]): T;
}
```

`Class<T>` is a type for classes whose instances match type `T`.

23.3.1 Example: creating instances

`Class<T>` enables us to write a generic version of `createPoint()`:

```
function createInstance<T>(TheClass: Class<T>, ...args: unknown[]): T {
  return new TheClass(...args);
}
```

`createInstance()` is used as follows:

```
class Person {
  constructor(public name: string) {}
}

const jane = createInstance(Person, 'Jane');
assertType<Person>(jane);
```

`createInstance()` is the `new` operator, implemented via a function.

23.3.2 Example: type-narrowing via `instanceof`

In line A, `instanceof` narrows the type of `arg`: Before, it is `unknown`. After, it is `T`.

```
function isInstance<T>(TheClass: Class<T>, arg: unknown): boolean {
  type _ = Assert<Equal<
    typeof arg, unknown
  >>;
  if (arg instanceof TheClass) { // (A)
    type _ = Assert<Equal<
      typeof arg, T
    >>;
    return true;
  }
  return false;
}
```

23.3.3 Example: casting with runtime checks

We can use `Class<T>` to implement casting:

```
function cast<T>(TheClass: Class<T>, value: unknown): T {
  if (!(value instanceof TheClass)) {
    throw new Error(`Not an instance of ${TheClass.name}: ${value}`)
  }
  return value;
}
```

With `cast()`, we can change the type of a value to something more specific. This is also safe at runtime, because we both statically change the type and perform a dynamic check. The following code provides an example:

```
function parseObject(jsonObjectStr: string): Object {
  const parsed = JSON.parse(jsonObjectStr);
  type _ = Assert<Equal<
    typeof parsed, any
```

```
  >>;
  return cast(Object, parsed);
}
```

23.3.4 Example: an assertion function

We can turn function `cast()` from the previous subsection into an assertion function:

```
/**
 * After invoking this function, the inferred type of `value` is `T`.
 */
export function throwIfNotInstance<T>(
  TheClass: Class<T>, value: unknown
): asserts value is T { // (A)
  if (!(value instanceof TheClass)) {
    throw new Error(`Not an instance of ${TheClass}: ${value}`);
  }
}
```

The return type (line A) makes `throwIfNotInstance()` an assertion function that narrows types:

```
const parsed = JSON.parse('[1, 2]');
type _1 = Assert<Equal<
  typeof parsed, any
>>;
throwIfNotInstance(Array, parsed);
type _2 = Assert<Equal<
  typeof parsed, Array<unknown>
>>;
```

23.3.5 Example: Maps that are type-safe at runtime

One use case for `Class<T>` and `cast()` is type-safe Maps:

```
class TypeSafeMap {
  #data = new Map<unknown, unknown>();
  get<T>(key: Class<T>) {
    const value = this.#data.get(key);
    return cast(key, value);
  }
  set<T>(key: Class<T>, value: T): this {
    cast(key, value); // runtime check
    this.#data.set(key, value);
    return this;
  }
  has(key: unknown) {
    return this.#data.has(key);
  }
}
```

The key of each entry in a `TypeSafeMap` is a class. That class determines the static type of the entry's value and is also used for checks at runtime.

This is `TypeSafeMap` in action:

```
const map = new TypeSafeMap();

map.set(RegExp, /abc/);

const re = map.get(RegExp);
assertType<RegExp>(re);

// Static and dynamic error!
assert.throws(
  // @ts-expect-error: Argument of type 'string' is not assignable
  // to parameter of type 'Date'.
  () => map.set(Date, 'abc')
);
```

23.3.6 Pitfall: `Class<T>` does not match abstract classes

Consider the following classes:

```
abstract class Shape {
}
class Circle extends Shape {
  // ···
}
```

`Class<T>` does not match the abstract class `Shape` (last line):

```
type Class<T> = new (...args: any[]) => T;

// @ts-expect-error: Type 'typeof Shape' is not assignable to
// type 'Class<Shape>'. Cannot assign an abstract constructor type
// to a non-abstract constructor type.
const shapeClasses1: Array<Class<Shape>> = [Circle, Shape];
```

Why is that? The rationale is that constructor type literals and construct signatures should only be used for values that can actually be `new`-invoked. If we want to `Class<T>` to match both abstract and concrete classes, we can use an *abstract construct signature*:

```
type Class<T> = abstract new (...args: any[]) => T;
const shapeClasses: Array<Class<Shape>> = [Circle, Shape];
```

There is once caveat – this type cannot be `new`-invoked:

```
function createInstance<T>(TheClass: Class<T>, ...args: unknown[]): T {
  // @ts-expect-error: Cannot create an instance of an abstract class.
  return new TheClass(...args);
}
```

However, the new `Class<T>` works well for all other use cases, including `instanceof`:

```
function isInstance<T>(TheClass: Class<T>, arg: unknown): boolean {
  type _ = Assert<Equal<
    typeof arg, unknown
  >>;
  if (arg instanceof TheClass) {
    type _ = Assert<Equal<
      typeof arg, T
    >>;
    return true;
  }
  return false;
}
```

Therefore, we can rename the old type for classes to `NewableClass<T>` – in case we need a class to be `new`-invokable:

```
type NewableClass<T> = new (...args: any[]) => T;
function createInstance<T>(TheClass: NewableClass<T>, ...args: unknown[]): T {
  return new TheClass(...args);
}
```

Chapter 24

Where are the remaining chapters?

You are reading a preview version of this book. You can either read all chapters online or you can buy the full version.