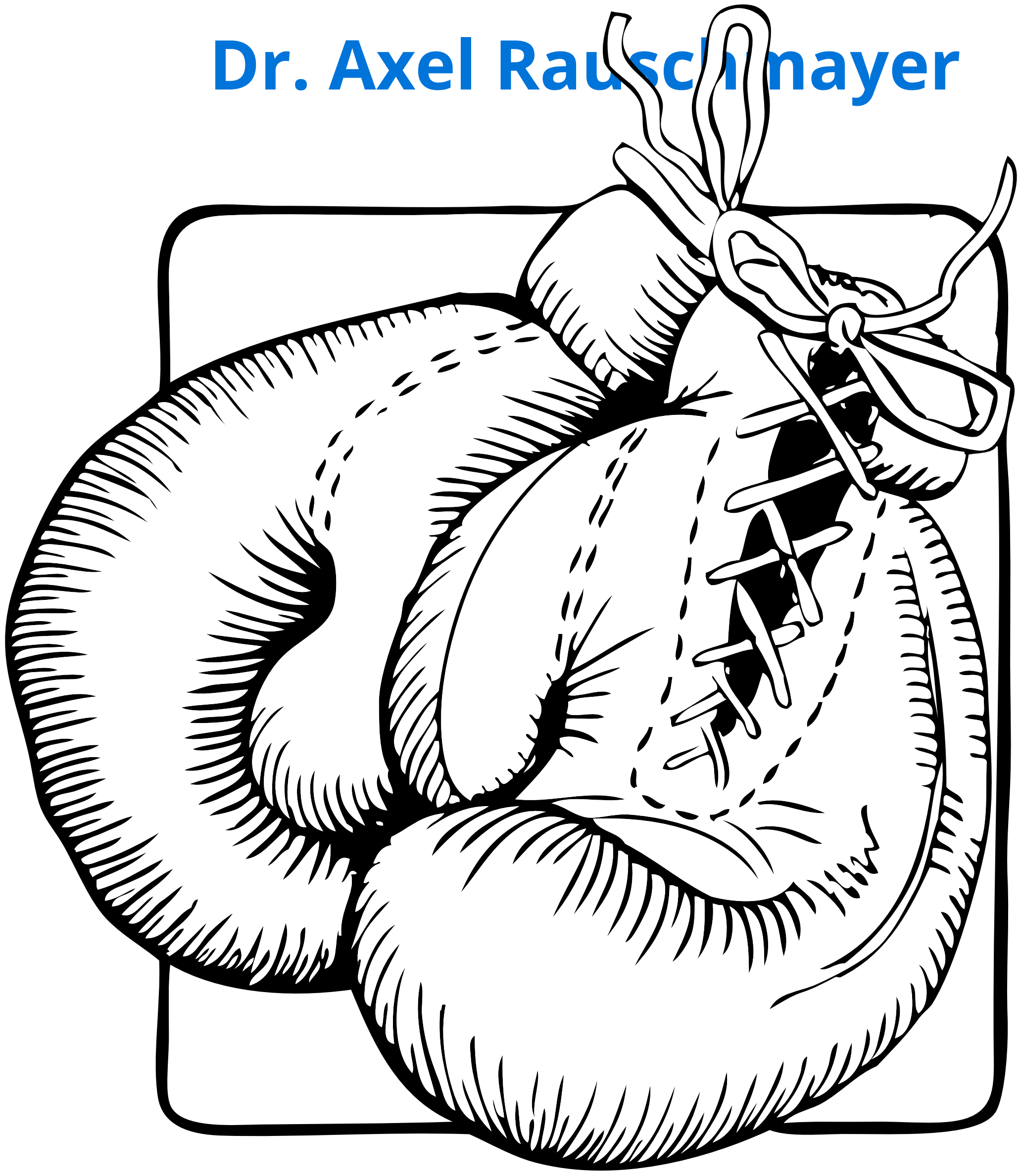
Dr. Axel Rauschmayer



Tackling TypeScript

Upgrading from JavaScript

Tackling TypeScript

Dr. Axel Rauschmayer

2020

exploringjs.com

Contents

Part I

Preliminaries

Chapter 1

About this book

Contents

1.1 Where is the homepage of this book?

The homepage of "Tackling TypeScript" is `exploringjs.com/tackling-ts/`

1.2 What is in this book?

This book consists of two parts:

- Part 1 is a quick start for TypeScript that teaches you the essentials quickly.
- Part 2 digs deeper into the language and covers many important topics in detail.

This book is not a reference, it is meant to complement the official TypeScript handbook.

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

1.3 What do I get for my money?

If you buy this book, you get:

- The current content in four DRM-free versions:

  - PDF file
  - ZIP archive with ad-free HTML
  - EPUB file
  - MOBI file
- Any future content that is added to this edition. How much I can add depends on the sales of this book.

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 (in which order, what to omit, etc.).

External content

Points to additional, external, content.

Git repository

Mentions a relevant Git repository.

Tip

Gives a tip.

Question

Asks and answers a question (think FAQ).

Warning

Warns about a pitfall, etc.

Details

Provides additional details, similar to a footnote.

1.7 Acknowledgements

People who contributed to this book are acknowledged in the chapters.

Chapter 2

Why TypeScript?

Contents

You can skip this chapter if you are already sure that you will learn and use TypeScript.

If you are still unsure – this chapter is my sales pitch.

2.1 The benefits of using TypeScript

2.1.1 More errors are detected *statically* (without running code)

While you are editing TypeScript code in an integrated development environment, you get warnings if you mistype names, call functions incorrectly, etc.

Consider the following two lines of code:

```
function func() {}
funcc();
```

For the second line, we get this warning:

```
Cannot find name 'funcc'. Did you mean 'func'?
```

Another example:

```
const a = 0;
const b = true;
const result = a + b;
```

This time, the error message for the last line is:

```
Operator '+' cannot be applied to types 'number' and 'boolean'.
```

2.1.2 Documenting parameters is good practice anyway

Documenting parameters of functions and methods is something that many people do, anyway:

```
/**
 * @param {number} num - The number to convert to string
 * @returns {string} `num`, converted to string
 */
function toString(num) {
  return String(num);
}
```

Specifying the types via `{number}` and `{string}` is not required, but the descriptions in English mention them, too.

If we use TypeScript's notation to document types, we get the added benefit of this information being checked for consistency:

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

2.1.3 TypeScript provides an additional layer of documentation

Whenever I migrate JavaScript code to TypeScript, I'm noticing an interesting phenomenon: In order to find the appropriate types for parameters for 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.

And I do indeed find it easier to understand TypeScript code bases than JavaScript code bases: 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.

2.1.4 Type definitions for JavaScript improve auto-completion

If there are type definitions for JavaScript code, then editors can use them to improve auto-completion.

An alternative to using TypeScript's syntax, is to provide all type information via JSDoc comments – like we did at the beginning of this chapter. In that case, TypeScript can also check code for consistency and generate type definitions. For more information, see chapter "Type Checking JavaScript Files" in the TypeScript handbook.

2.1.5 TypeScript makes refactorings safer

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.

2.1.6 TypeScript can compile new features to older code

TypeScript tends to quickly support ECMAScript stage 4 features (such features are scheduled to be included in the next ECMAScript version). When we compile to JavaScript, the compiler option `--target` lets us specify the ECMAScript version that the output is compatible with. Then any incompatible feature (that was introduced later) will be compiled to equivalent, compatible code.

Note that this kind of support for older ECMAScript versions does not require TypeScript or static typing: The JavaScript compiler Babel does it too, but it compiles JavaScript to JavaScript.

2.2 The downsides of using TypeScript

- It is an added layer on top of JavaScript: more complexity, more things to learn, etc.
- It introduces a compilation step when writing code.
- npm packages can only be used if they have static type definitions.
  - These days, many 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.
- Getting static types right is occasionally difficult. My recommendation here is to keep things as simple as possible – for example: Don't overdo generics and type variables.

2.3 TypeScript myths

2.3.1 TypeScript code is heavyweight

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

```
function selectionSort(arr: number[]) { // (A)
  for (let i=0; i<arr.length; i++) {
    const minIndex = findMinIndex(arr, i);
```

```
      [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]]; // swap
    }
  }

  function findMinIndex(arr: number[], startIndex: number) { // (B)
    let minValue = arr[startIndex];
    let minIndex = startIndex;
    for (let i=startIndex+1; i < arr.length; i++) {
      const curValue = arr[i];
      if (curValue < minValue) {
        minValue = curValue;
        minIndex = i;
      }
    }
    return minIndex;
  }

  const arr = [4, 2, 6, 3, 1, 5];
  selectionSort(arr);
  assert.deepEqual(
    arr, [1, 2, 3, 4, 5, 6]);
```

The only locations where this TypeScript code is different from JavaScript code, are line A and line B.

There are a variety of styles in which TypeScript is written:

- In an object-oriented programming (OOP) style with classes and OOP patterns
- In a functional programming (FP) style with functional patterns
- In a mix of OOP and FP
- And so on

2.3.2 TypeScript is an attempt to replace JavaScript with C# or Java

Initially, TypeScript did invent a few language constructs of its own (e.g. enums). But since ECMAScript 6, it mostly stuck with being a strict superset of JavaScript.

My impression is that the TypeScript team likes JavaScript and doesn't want to replace it with something "better" (which is the goal of, e.g., Dart). They do want to make it possible to statically type as much JavaScript code as possible. Many new TypeScript features are driven by that desire.

Chapter 3

Free resources on TypeScript

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 "JavaScript for impatient programmers" which is free to read online. Some of the "Further reading" sections at the ends of chapters refer to this book.

Books on TypeScript:

- The "TypeScript Handbook" is a good reference for the language. I see "Tackling TypeScript" as complementary to that book.
- "TypeScript Deep Dive" by Basarat Ali Syed

More material:

- The "TypeScript Language Specification" explains the lower levels of the language.
- Marius Schulz publishes blog posts on TypeScript and the email newsletter "TypeScript Weekly".
- 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

Getting started with TypeScript

Chapter 4

How does TypeScript work? The bird's eye view

Contents

This chapter gives the bird's eye view of how TypeScript works: What is the structure of a typical TypeScript project? What is compiled and how? How can we use IDEs to write TypeScript?

4.1 The structure of TypeScript projects

This is one possible file structure for TypeScript projects:

```
typescript-project/
  dist/
  ts/
    src/
      main.ts
      util.ts
    test/
      util_test.ts
  tsconfig.json
```

Explanations:

- Directory `ts/` contains the TypeScript files:
  - Subdirectory `ts/src/` contains the actual code.
  - Subdirectory `ts/test/` contains tests for the code.
- Directory `dist/` is where the output of the compiler is stored.
- The TypeScript compiler compiles TypeScript files in `ts/` to JavaScript files in `dist/`. For example:
  - `ts/src/main.ts` is compiled to `dist/src/main.js` (and possibly other files)
- `tsconfig.json` is used to configure the TypeScript compiler.

4.1.1 `tsconfig.json`

The contents of `tsconfig.json` look as follows:

```
{
  "compilerOptions": {
    "rootDir": "ts",
    "outDir": "dist",
    "module": "commonjs",
    ···
  }
}
```

We have specified that:

- The root directory of the TypeScript code is `ts/`.
- The directory where the TypeScript compiler saves its output is `dist/`.
- The module format of the output files is CommonJS.

4.2 Programming TypeScript via an integrated development environment (IDE)

Two popular IDEs for JavaScript are:

- *Visual Studio Code* (free)
- *WebStorm* (for purchase)

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

One important fact to be aware of is that Visual Studio Code processes TypeScript source code in two independent ways:

- Checking open files for errors: This is done via a so-called *language server*. Language servers exist independently of particular editors and provide Visual Studio Code with language-related services: detecting errors, refactorings, auto-completions, etc. Communication with servers happens via a protocol that is based on JSON-RPC (*RPC* stands for *remote procedure calls*). The independence provided by that protocol means that servers can be written in almost any programming language.
  - Important fact to remember: The language server only lists errors for currently open files and doesn't compile TypeScript, it only analyzes it statically.

- *Building* (compiling TypeScript files to JavaScript files): Here, we have two choices.
  - We can run a build tool via an external command line. For example, the TypeScript compiler `tsc` has a `--watch` mode that watches input files and compiles them to output files whenever they change. As a consequence, whenever we save a TypeScript file in the IDE, we immediately get the corresponding output file(s).
  - We can run `tsc` from within Visual Studio Code. In order to do so, it must be installed either inside project that we are currently working on or globally (via the Node.js package manager npm).

  With building, we get a complete list of errors. For more information on compiling TypeScript from within Visual Studio Code, see the official documentation for that IDE.

4.3 Other files produced by the TypeScript compiler

Given a TypeScript file `main.ts`, the TypeScript compiler can produce several kinds of artifacts. The most common ones are:

- JavaScript file: `main.js`
- Declaration file: `main.d.ts` (contains type information; think `.ts` file minus the JavaScript code)
- Source map file: `main.js.map`

TypeScript is often not delivered via `.ts` files, but via `.js` files and `.d.ts` files:

- The JavaScript code contains the actual functionality and can be consumed via plain JavaScript.
- The declaration files help programming editors with auto-completion and similar services. This information enables plain JavaScript to be consumed via TypeScript. However, we even profit from it if we work with plain JavaScript because it gives us better auto-completion and more.

A source map specifies for each part of the output code in `main.js`, which part of the input code in `main.ts` produced it. Among other things, this information enables runtime environments to execute JavaScript code, while showing the line numbers of the TypeScript code in error messages.

4.3.1 In order to use npm packages from TypeScript, we need type information

The npm registry is a huge repository of JavaScript code. If we want to use a JavaScript package from TypeScript, we need type information for it:

- The package itself may include `.d.ts` files or even the complete TypeScript code.
- If it doesn't, we may still be able to use it: DefinitelyTyped is a repository of declaration files that people have written for plain JavaScript packages.

The declaration files of DefinitelyTyped reside in the `@types` namespace. Therefore, if we need a declaration file for a package such as `lodash`, we have to install the package `@types/lodash`.

4.4 Using the TypeScript compiler for plain JavaScript files

The TypeScript compiler can also process plain JavaScript files:

- With the option `--allowJs`, the TypeScript compiler copies JavaScript files in the input directory over to the output directory. Benefit: When migrating from JavaScript to TypeScript, we can start with a mix of JavaScript and TypeScript files and slowly convert more JavaScript files to TypeScript.
- With the option `--checkJs`, the compiler additionally type-checks JavaScript files (`--allowJs` must be on for this option to work). It does so as well as it can, given the limited information that is available. Which files are checked can be configured via comments inside them:
  - Explicit excluding: If a JavaScript file contains the comment `// @ts-nocheck`, it will not be type-checked.
  - Explicit including: Without `--checkJs`, the comment `// @ts-check` can be used to type-check individual JavaScript files.
- The TypeScript compiler uses static type information that is specified via JSDoc comments (see below for an example). If we are thorough, we can fully statically type plain JavaScript files and even derive declaration files from them.
- With the option `--noEmit`, the compiler does not produce any output, it only type-checks files.

This is an example of a JSDoc comment that provides static type information for a function `add()`:

```
/**
 * @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;
}
```

More information: Type-Checking JavaScript Files in the TypeScript Handbook.

Chapter 5

Trying out TypeScript

Contents

This chapter gives tips for quickly trying out TypeScript.

5.1 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. It can also execute the result 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/MYewdgzgLgBFDuBLYBTGBeGA
KAHgLhmgCdEwBzASgwD4YcYBqOgbgChXRIQAbFAOm4gyWBMhRYA5AEMARsAkUKzIA
```

5.2 TS Node

TS Node is a TypeScript version of Node.js. Its use cases are:

- TS Node provides a REPL (command line) for TypeScript:

```
$ ts-node
> const twice = (x: string) => x + x;
> twice('abc')
'abcabc'
```

```
> twice(123)
Error TS2345: Argument of type '123' is not assignable
to parameter of type 'string'.
```

- TS Node enables some JavaScript tools to directly execute TypeScript code. It automatically compiles TypeScript code to JavaScript code and passes it on to the tools, without us having to do anything. The following shell command demonstrates how that works with the JavaScript unit test framework Mocha:

  ```
  mocha --require ts-node/register --ui qunit testfile.ts
  ```

Use `npx ts-node` to run the REPL without installing it.

Chapter 6

Notation used in this book

Contents

This chapter explains functionality that is used in the code examples, but not part of TypeScript proper.

6.1 Test assertions (dynamic)

The code examples shown in this book are tested automatically via unit tests. Expected results of operations are checked via the following assertion functions from the Node.js module `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 {strict as assert} from 'assert';

assert.equal(3 + ' apples', '3 apples');

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

assert.throws(
  () => eval('null.myProperty'),
  TypeError);
```

The import statement in the first line makes use of strict assertion mode (which uses ===, not ==). It is usually omitted in code examples.

6.2 Type assertions (static)

You'll also see static type assertions.

`%inferred-type` is just a comment in normal TypeScript and describes the type that TypeScript infers for the following line:

```
// %inferred-type: number
let num = 123;
```

`@ts-expect-error` suppresses static errors in TypeScript. In this book, the suppressed error is always mentioned. That is neither required in plain TypeScript, nor does it do anything there.

```
assert.throws(
  // @ts-expect-error: Object is possibly 'null'. (2531)
  () => null.myProperty,
  TypeError);
```

Note that we previously needed `eval()` in order to not be warned by TypeScript.

Chapter 7

The essentials of TypeScript

Contents

This chapter explains the essentials of TypeScript.

7.1 What you'll learn

After reading this chapter, you should be able to understand the following TypeScript code:

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

7.2 Specifying the comprehensiveness of type checking

There are many ways in which the TypeScript compiler can be configured. One important group of options controls how thoroughly the compiler checks TypeScript code. The maximum setting is activated via `--strict` and I recommend to always use it. It makes programs slightly harder to write, but we also gain the full benefits of static type checking.

That's everything about `--strict` you need to know for now

Read on if you want to know more details.

Setting `--strict` to `true`, sets all of the following options to `true`:

- `--noImplicitAny`: If TypeScript can't infer a type, we must specify it. This mainly applies to parameters of functions and methods: With this settings, we must annotate them.
- `--noImplicitThis`: Complain if the type of `this` isn't clear.
- `--alwaysStrict`: Use JavaScript's strict mode whenever possible.

- `--strictNullChecks`: `null` is not part of any type (other than its own type, `null`) and must be explicitly mentioned if it is a acceptable value.
- `--strictFunctionTypes`: enables stronger checks for function types.
- `--strictPropertyInitialization`: Properties in class definitions must be initialized, unless they can have the value `undefined`.

We will see more compiler options later in this book, when we get to creating npm packages and web apps with TypeScript. The TypeScript handbook has comprehensive documentation on them.

7.3 Types in TypeScript

In this chapter, a type is simply a set of values. The JavaScript language (not TypeScript!) has only eight types:

1. Undefined: the set with the only element `undefined`
2. Null: the set with the only element `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)

All of these types are *dynamic*: we can use them at runtime.

TypeScript brings an additional layer to JavaScript: *static types*. These only exist when compiling or type-checking source code. Each storage location (variable, property, etc.) has a static type that predicts its dynamic values. Type checking ensures that these predictions come true.

And there is a lot that can be checked *statically* (without running the code). If, for example the parameter `num` of a function `toString(num)` has the static type `number`, then the function call `toString('abc')` is illegal, because the argument `'abc'` has the wrong static type.

7.4 Type annotations

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

There are two type annotations in the previous function declaration:

- Parameter `num`: colon followed by `number`
- Result of `toString()`: colon followed by `string`

Both `number` and `string` are *type expressions* that specify the types of storage locations.

7.5 Type inference

Often, TypeScript can *infer* a static type if there is no type annotation. For example, if we omit the return type of `f()`, TypeScript infers that it is `string`:

```
// %inferred-type: (num: number) => string
function toString(num: number) {
  return String(num);
}
```

Type inference is not guesswork: It follows clear rules (similar to arithmetic) for deriving types where they haven't been specified explicitly. In this case, the return statement applies a function `String()` that maps arbitrary values to strings, to a value `num` of type `number` and returns the result. That's why the inferred return type is `string`.

If the type of a location is neither explicitly specified nor inferrable, TypeScript uses the type `any` for it. This is the type of all values and a wildcard, in that we can do everything if a value has that type.

With `--strict`, `any` is only allowed if we use it explicitly. In other words: Every location must have an explicit or inferred static type. In the following example, parameter `num` has neither and we get a compile-time error:

```
// @ts-expect-error: Parameter 'num' implicitly has an 'any' type. (7006)
function toString(num) {
  return String(num);
}
```

7.6 Specifying types via type expressions

The type expressions after the colons of type annotations range from simple to complex and are created as follows.

Basic types are valid type expressions:

- Static types for JavaScript's dynamic types:
  - `undefined`, `null`
  - `boolean`, `number`, `bigint`, `string`
  - `symbol`
  - `object`.
- TypeScript-specific types:
  - `Array` (not technically a type in JavaScript)
  - `any` (the type of all values)
  - Etc.

There are many ways of combining basic types to produce new, *compound types*. For example, via *type operators* that combine types similarly to how the set operators *union* (∪) and *intersection* (∩) combine sets. We'll see how to do that soon.

7.7 The two language levels: dynamic vs. static

TypeScript has two language levels:

- The *dynamic level* is managed by JavaScript and consists of code and values, at runtime.
- The *static level* is managed by TypeScript (excluding JavaScript) and consists of static types, at compile time.

We can see these two levels in the syntax:

```
const undef: undefined = undefined;
```

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

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

Note that the same syntax, `undefined`, means different things depending on whether it is used at the dynamic level or at the static level.

> **Try to develop an awareness of the two language levels**
>
> That helps considerably with making sense of TypeScript.

7.8 Type aliases

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

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

7.9 Typing Arrays

Arrays play two roles in JavaScript (either one or both):

- List: All elements have the same type. The length of the Array varies.
- Tuple: The length of the Array is fixed. The elements generally don't have the same type.

7.9.1 Arrays as lists

There are two ways to express the fact that the Array `arr` is used as a list whose elements are all numbers:

```
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 actually have to help it, because with an empty Array, it can't determine the type of the elements.

We'll get back to the angle brackets notation (`Array<number>`) later.

7.9.2 Arrays as tuples

If we store a two-dimensional point in an Array, then we are using that Array as a tuple. That looks as follows:

```
let point: [number, number] = [7, 5];
```

The type annotation is needed for Arrays-as-tuples because, for Array literals, TypeScript infers list types, not tuple types:

```
// %inferred-type: number[]
let point = [7, 5];
```

Another example for tuples is the result of `Object.entries(obj)`: an Array with one [key, value] pair for each property of `obj`.

```
// %inferred-type: [string, number][]
const entries = Object.entries({ a: 1, b: 2 });

assert.deepEqual(
  entries,
  [[ 'a', 1 ], [ 'b', 2 ]]);
```

The inferred type is an Array of tuples.

7.10 Function types

This is an example of a function type:

```
(num: number) => string
```

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

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

Normally, we must specify parameter types for functions. But in this case, the type of `num` in line B can be inferred from the function type in line A and we can omit it:

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

If we omit the type annotation for `toString`, TypeScript infers a type from the arrow function:

```
// %inferred-type: (num: number) => string
const toString = (num: number) => String(num);
```

This time, `num` must have a type annotation.

7.10.1 A more complicated example

The following example is more complicated:

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

We are using a function type to describe the parameter `callback` of `stringify123()`. Due to this type annotation, TypeScript rejects the following function call.

```
// @ts-expect-error: Argument of type 'NumberConstructor' is not
// assignable to parameter of type '(num: number) => string'.
//   Type 'number' is not assignable to type 'string'.(2345)
stringify123(Number);
```

But it accepts this function call:

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

7.10.2 Return types of function declarations

TypeScript can usually infer the return types of functions, but specifying them explicitly is allowed and occasionally useful (at the very least, it doesn't do any harm).

For `stringify123()`, specifying a return type is optional and looks like this:

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

7.10.2.1 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 '"abc"' is not assignable to type 'void'. (2322)
  return 'abc';
}
```

7.10.3 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).

7.10.3.1 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];
}
```

7.10.4 Rest parameters

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

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

7.11 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).

7.11.1 By default, `undefined` and `null` are not included in types

In many programming languages, `null` is part of all object types. For example, whenever the type of a variable is `String` in Java, we can set it to `null` and Java won't complain.

Conversely, in TypeScript, `undefined` and `null` are handled by separate, disjoint types. We need union types such as `undefined|string` and `null|string`, if we want to allow them:

```
let maybeNumber: null|number = null;
maybeNumber = 123;
```

Otherwise, we get an error:

```
// @ts-expect-error: Type 'null' is not assignable to type 'number'. (2322)
let maybeNumber: number = null;
maybeNumber = 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;
```

7.11.2 Making omissions explicit

Recall this function from earlier:

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

Let's rewrite `stringify123()` so that parameter `callback` isn't optional anymore: If a caller doesn't want to provide a function, they must explicitly pass `null`. The result looks as follows.

```
function stringify123(
  callback: null | ((num: number) => string)) {
  const num = 123;
  if (callback === null) { // (A)
    callback = String;
  }
  return callback(num); // (B)
}

assert.equal(
  stringify123(null),
  '123');

// @ts-expect-error: Expected 1 arguments, but got 0. (2554)
assert.throws(() => stringify123());
```

Once again, we have to handle the case of `callback` not being a function (line A) before we can make the function call in line B. If we hadn't done so, TypeScript would have reported an error in that line.

7.12 Optional vs. default value vs. `undefined|T`

The following three parameter declarations are quite similar:

- Parameter is optional: `x?: number`
- Parameter has a default value: `x = 456`
- Parameter has a union type: `x: undefined | number`

If the parameter is optional, it can be omitted. In that case, it has the value `undefined`:

```
function f1(x?: number) { return x }

assert.equal(f1(123), 123); // OK
assert.equal(f1(undefined), undefined); // OK
assert.equal(f1(), undefined); // can omit
```

If the parameter has a default value, that value is used when the parameter is either omitted or set to `undefined`:

```
function f2(x = 456) { return x }
```

```
assert.equal(f2(123), 123); // OK
assert.equal(f2(undefined), 456); // OK
assert.equal(f2(), 456); // can omit
```

If the parameter has a union type, it can’t be omitted, but we can set it to `undefined`:

```
function f3(x: undefined | number) { return x }

assert.equal(f3(123), 123); // OK
assert.equal(f3(undefined), undefined); // OK

// @ts-expect-error: Expected 1 arguments, but got 0. (2554)
f3(); // can’t omit
```

7.13 Typing objects

Similarly to Arrays, objects play two roles in JavaScript (that are occasionally mixed):

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

We are ignoring objects-as-dictionaries in this chapter – they are covered in §15.4.5 “Index signatures: objects as dicts”. As an aside, Maps are usually a better choice for dictionaries, anyway.

7.13.1 Typing objects-as-records via interfaces

Interfaces describe objects-as-records. For example:

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

We can also separate members via commas:

```
interface Point {
  x: number,
  y: number,
}
```

7.13.2 TypeScript’s structural typing vs. nominal typing

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

```
interface 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.

7.13.3 Object literal types

Object literal types are anonymous interfaces:

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

One benefit of object literal types is that they can be used inline:

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

7.13.4 Optional properties

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

```
interface Person {
  name: string;
  company?: string;
}
```

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

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

7.13.5 Methods

Interfaces can also contain methods:

```
interface 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:

```
interface HasMethodDef {
  simpleMethod(flag: boolean): void;
}
interface HasFuncProp {
  simpleMethod: (flag: boolean) => void;
}

const objWithMethod: HasMethodDef = {
  simpleMethod(flag: boolean): void {},
};
const objWithMethod2: HasFuncProp = objWithMethod;

const objWithOrdinaryFunction: HasMethodDef = {
  simpleMethod: function (flag: boolean): void {},
};
const objWithOrdinaryFunction2: HasFuncProp = objWithOrdinaryFunction;

const objWithArrowFunction: HasMethodDef = {
  simpleMethod: (flag: boolean): void => {},
};
const objWithArrowFunction2: HasFuncProp = objWithArrowFunction;
```

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

7.14 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
  ```

- Parameterized 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
```

7.14.1 Example: a container for values

```
// Factory for types
interface 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.

7.15 Example: a type-parameterized class

Classes can have type parameters, too:

```
class SimpleStack<T> {
  #data: Array<T> = [];
  push(x: T): void {
    this.#data.push(x);
  }
  pop(): T {
    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 `T`. Single uppercase letters such as `T` are often used for type parameters.

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');
```

7.15.1 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:

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

7.15.2 Type variables for functions

Functions (and methods) can introduce type variables, too:

```
function id<T>(x: T): T {
  return x;
}
```

We use this function as follows.

```
// %inferred-type: number
const num1 = id<number>(123);
```

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

```
// %inferred-type: 123
const num2 = id(123);
```

Note that TypeScript inferred the type `123`, which is a set with one number and more specific than the type `number`.

7.15.3 A more complicated function example

```
// %inferred-type: <T>(len: number, elem: T) => T[]
function fillArray<T>(len: number, elem: T) {
  return new Array<T>(len).fill(elem);
}
```

The type variable `T` appears three 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 also used as a type argument for the constructor `Array()`.

The return type of `fillArray()` is inferred, but we also could have specified it explicitly.

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

```
// %inferred-type: string[]
const arr1 = fillArray<string>(3, '*');
assert.deepEqual(
  arr1, ['*', '*', '*']);

// %inferred-type: string[]
const arr2 = fillArray(3, '*'); // (A)
```

7.16 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, array: T[]) => 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.
- 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
  - `array` with elements of type `T`

Chapter 8

Creating CommonJS-based npm packages via TypeScript

Contents

This chapter describes how to use TypeScript to create packages for the package manager npm that are based on the CommonJS module format.

GitHub repository: `ts-demo-npm-cjs`

In this chapter, we are exploring the repository `ts-demo-npm-cjs` which can be downloaded on GitHub. (I deliberately have not published it as a package to npm.)

8.1 Required knowledge

You should be roughly familiar with:

- *CommonJS modules* – a module format that originated in, and was designed for, server-side JavaScript. It was popularized by the server-side JavaScript platform *Node.js*. CommonJS modules preceded JavaScript's built-in ECMAScript modules and are still much used and very well supported by tooling (IDEs, built tools, etc.).
- TypeScript's modules – whose syntax is based on ECMAScript modules. However, they are often compiled to CommonJS modules.
- npm packages – directories with files that are installed via the npm package manager. They can contain CommonJS modules, ECMAScript modules, and various other files.

8.2 Limitations

In this chapter, we are using what TypeScript currently supports best:

- All our TypeScript code is compiled to CommonJS modules with the filename extension `.js`.
- All external imports are CommonJS modules, too.

Especially on Node.js, TypeScript currently doesn't really support ECMAScript modules and filename extensions other than `.js`.

8.3 The repository `ts-demo-npm-cjs`

This is how the repository `ts-demo-npm-cjs` is structured:

```
ts-demo-npm-cjs/
  .gitignore
  .npmignore
  dist/   (created on demand)
  package.json
  ts/
    src/
      index.ts
    test/
      index_test.ts
  tsconfig.json
```

Apart from the `package.json` for the package, the repository contains:

- `ts/src/index.ts`: the actual code of the package
- `ts/test/index_test.ts`: a test for `index.ts`
- `tsconfig.json`: configuration data for the TypeScript compiler

`package.json` contains scripts for compiling:

- Input: directory `ts/` (TypeScript code)
- Output: directory `dist/` (CommonJS modules; the directory doesn't yet exist in the repository)

This is where the compilation results for the two TypeScript files are put:

```
ts/src/index.ts       --> dist/src/index.js
ts/test/index_test.ts --> dist/test/index_test.js
```

8.4 .gitignore

This file lists the directories that we don't want to check into git:

```
node_modules/
dist/
```

Explanations:

- `node_modules/` is set up via `npm install`.
- The files in `dist/` are created by the TypeScript compiler (more on that later).

8.5 .npmignore

When it comes to which files should and should not be uploaded to the npm registry, we have different needs than we did for git. Therefore, in addition to `.gitignore`, we also need the file `.npmignore`:

```
ts/
```

The two differences are:

- We want to upload the results of compiling TypeScript to JavaScript (directory `dist/`).
- We don't want to upload the TypeScript source files (directory `ts/`).

Note that npm ignores the directory `node_modules/` by default.

8.6 package.json

`package.json` looks like this:

```
{
  ···
  "type": "commonjs",
  "main": "./dist/src/index.js",
  "types": "./dist/src/index.d.ts",
  "scripts": {
    "clean": "shx rm -rf dist/*",
    "build": "tsc",
    "watch": "tsc --watch",
    "test": "mocha --ui qunit",
    "testall": "mocha --ui qunit dist/test",
    "prepack": "npm run clean && npm run build"
  },
  "// devDependencies": {
    "@types/node": "Needed for unit test assertions (assert.equal() etc.)",
    "shx": "Needed for development-time package.json scripts"
  },
  "devDependencies": {
    "@types/lodash": "···",
    "@types/mocha": "···",
    "@types/node": "···",
    "mocha": "···",
    "shx": "···"
  },
  "dependencies": {
    "lodash": "···"
  }
}
```

Let's take a look at the properties:

- `type`: The value `"commonjs"` means that `.js` files are interpreted as CommonJS modules.
- `main`: If there is a so-called *bare import* that only mentions the name of the current package, then this is the module that will be imported.
- `types` points to a declaration file with all the type definitions for the current package.

The next two subsections cover the remaining properties.

8.6.1 Scripts

Property `scripts` defines various commands that can be invoked via `npm run`. For example, the script `clean` is invoked via `npm run clean`. The previous `package.json` contains the following scripts:

- `clean` uses the cross-platform package `shx` to delete the compilation results via its implementation of the Unix shell command `rm`. `shx` supports a variety of shell commands with the benefit of not needing a separate package for each command we may want to use.

- `build` and `watch` use the TypeScript compiler `tsc` to compile the TypeScript files according to `tsconfig.json`. `tsc` must be installed globally or locally (inside the current package), usually via the npm package `typescript`.

- `test` and `testall` use the unit test framework Mocha to run one test or all tests.

- `prepack`: This script is run run before a tarball is packed (due to `npm pack`, `npm publish`, or an installation from git).

Note that when we are using an IDE, we don't need the scripts `build` and `watch` because we can let the IDE build the artifacts. But they are needed for the script `prepack`.

8.6.2 `dependencies` vs. `devDependencies`

`dependencies` should only contain the packages that are needed when importing a package. That excludes packages that are used for running tests etc.

Packages whose names start with `@types/` provide TypeScript type definitions for packages that don't have any. Without the former, we can't use the latter. Are these normal dependencies or dev dependencies? It depends:

- If the type definitions of our package refer to type definitions in another package, that package is a normal dependency.
- Otherwise, the package is only needed during development time and a dev dependency.

8.6.3 More information on `package.json`

- "Awesome npm scripts" has tips for writing cross-platform scripts.
- The npm docs for `package.json` explain various properties of that file.
- The npm docs for `scripts` explain the `package.json` property `scripts`.

8.7 `tsconfig.json`

```
{
  "compilerOptions": {
    "rootDir": "ts",
    "outDir": "dist",
    "target": "es2019",
    "lib": [
      "es2019"
    ],
    "module": "commonjs",
    "esModuleInterop": true,
    "strict": true,
    "declaration": true,
    "sourceMap": true
  }
}
```

- `rootDir`: Where are our TypeScript files located?
- `outDir`: Where should the compilation results be put?
- `target`: What is the targeted ECMAScript version? If the TypeScript code uses a feature that is not supported by the targeted version, then it is compiled to equivalent code that only uses supported features.
- `lib`: What platform features should TypeScript be aware of? Possibilities include the ECMAScript standard library and the DOM of browsers. The Node.js API is supported differently, via the package `@types/node`.
- `module`: Specifies the format of the compilation output.

The remaining options are explained by the official documentation for `tsconfig.json`.

8.8 TypeScript code

8.8.1 `index.ts`

This file provides the actual functionality of the package:

```
import endsWith from 'lodash/endsWith';

export function removeSuffix(str: string, suffix: string) {
  if (!endsWith(str, suffix)) {
    throw new Error(JSON.stringify(suffix)} + ' is not a suffix of ' +
      JSON.stringify(str));
  }
  return str.slice(0, -suffix.length);
}
```

It uses function `endsWith()` of the library Lodash. That's why Lodash is a normal dependency – it is needed at runtime.

8.8.2 `index_test.ts`

This file contains a unit test for `index.ts`:

```
import { strict as assert } from 'assert';
import { removeSuffix } from '../src/index';

test('removeSuffix()', () => {
  assert.equal(
    removeSuffix('myfile.txt', '.txt'),
    'myfile');
  assert.throws(() => removeSuffix('myfile.txt', 'abc'));
});
```

We can run the test like this:

```
npm t dist/test/index_test.js
```

- The npm command `t` is an abbreviation for the npm command `test`.
- The npm command `test` is an abbreviation for `run test` (which runs the script `test` from `package.json`).

As you can see, we are running the compiled version of the test (in directory `dist/`), not the TypeScript code.

For more information on the unit test framework Mocha, see its homepage.

Chapter 9

Creating web apps via TypeScript and webpack

Contents

This chapter describes how to create web apps via TypeScript and webpack. We will only be using the DOM API, not a particular frontend framework.

GitHub repository: `ts-demo-webpack`

The repository `ts-demo-webpack` that we are working with in this chapter, can be downloaded from GitHub.

9.1 Required knowledge

You should be roughly familiar with:

- npm
- webpack

9.2 Limitations

In this chapter, we stick with what is best supported by TypeScript: CommonJS modules, bundled as script files.

9.3 The repository `ts-demo-webpack`

This is how the repository `ts-demo-webpack` is structured:

```
ts-demo-webpack/
  build/   (created on demand)
  html/
    index.html
  package.json
  ts/
    src/
      main.ts
  tsconfig.json
  webpack.config.js
```

The web app is built as follows:

- Input:
  - The TypeScript files in `ts/`
  - All JavaScript code that is installed via npm and imported by the TypeScript files
  - The HTML files in `html/`
- Output – directory `build/` with the complete web app:
  - The TypeScript files are compiled to JavaScript code, combined with the npm-installed JavaScript and written to the script file `build/main-bundle.js`. This process is called *bundling* and `main-bundle.js` is a bundle file.
  - Each HTML file is copied to `build/`.

Both output tasks are handled by webpack:

- Copying the files in `html/` to `build/` is done via the webpack *plugin* `copy-webpack-plugin`.

- This chapter explores two different workflows for bundling:

  - Either webpack directly compiles TypeScript files into the bundle, with the help of the *loader* `ts-loader`.
  - Or we compile the TypeScript files ourselves, to Javascript files in the directory `dist/` (like we did in the previous chpater). Then webpack doesn't need a loader and only bundles JavaScript files.

  Most of this chapter is about using webpack with `ts-loader`. At the end, we briefly look at the other workflow.

9.4 package.json

package.json contains metadata for the project:

```
{
  "private": true,
  "scripts": {
    "tsc": "tsc",
    "tscw": "tsc --watch",
    "wp": "webpack",
    "wpw": "webpack --watch",
    "serve": "http-server build"
  },
  "dependencies": {
    "@types/lodash": "···",
    "copy-webpack-plugin": "···",
    "http-server": "···",
    "lodash": "···",
    "ts-loader": "···",
    "typescript": "···",
    "webpack": "···",
    "webpack-cli": "···"
  }
}
```

The properties work as follows:

- `"private": true` means that npm doesn't complain if we don't provide a package name and a package version.
- Scripts:
  - `tsc, tscw`: These scripts invoke the TypeScript compiler directly. We don't need them if we use webpack with `ts-loader`. However, they are useful if we use webpack without `ts-loader` (as demonstrated at the end of this chapter).
  - `wp`: runs webpack once, compile everything.
  - `wpw`: runs webpack in watch mode, where it watches the input files and only compiles files that change.
  - `serve`: runs the server `http-server` and serves the directory `build/` with the fully assembled web app.
- Dependencies:
  - Four packages related to webpack:
    * `webpack`: the core of webpack
    * `webpack-cli`: a command line interface for the core
    * `ts-loader`: a *loader* for `.ts` files that compiles them to JavaScript
    * `copy-webpack-plugin`: a *plugin* that copies files from one location to another one
  - Needed by `ts-loader`: `typescript`
  - Serves the web app: `http-server`
  - Library plus type definitions that the TypeScript code uses: `lodash`, `@types/lodash`

9.5 `webpack.config.js`

This is how we configure webpack:

```
const path = require('path');
const CopyWebpackPlugin = require('copy-webpack-plugin');

module.exports = {
  ...
  entry: {
    main: "./ts/src/main.ts",
  },
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: "[name]-bundle.js",
  },
  resolve: {
    // Add ".ts" and ".tsx" as resolvable extensions.
    extensions: [".ts", ".tsx", ".js"],
  },
  module: {
    rules: [
      // all files with a `.ts` or `.tsx` extension will be handled by `ts-loader`
      { test: /\.tsx?$/, loader: "ts-loader" },
    ],
  },
  plugins: [
    new CopyWebpackPlugin([
      {
        from: './html',
      }
    ]),
  ],
};
```

Properties:

- `entry`: An *entry point* is the file where webpack starts collecting the data for an output bundle. First it adds the entry point file to the bundle, then the imports of the entry point, then the imports of the imports, etc. The value of property `entry` is an object whose property keys specify names of entry points and whose property values specify paths of entry points.
- `output` specifies the path of the output bundle. `[name]` is mainly useful when there are multiple entry points (and therefore multiple output bundles). It is replaced with the name of the entry point when assembling the path.
- `resolve` configures how webpack converts *specifiers* (IDs) of modules to locations of files.
- `module` configures *loaders* (plugins that process files) and more.

- `plugins` configures *plugins* which can change and augment webpack's behavior in a variety of ways.

For more information on configuring webpack, see the webpack website.

9.6 `tsconfig.json`

This file configures the TypeScript compiler:

```json
{
  "compilerOptions": {
    "rootDir": "ts",
    "outDir": "dist",
    "target": "es2019",
    "lib": [
      "es2019",
      "dom"
    ],
    "module": "commonjs",
    "esModuleInterop": true,
    "strict": true,
    "sourceMap": true
  }
}
```

The option `outDir` is not needed if we use webpack with `ts-loader`. However, we'll need it if we use webpack without a loader (as explained later in this chapter).

9.7 `index.html`

This is the HTML page of the web app:

```html
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <title>ts-demo-webpack</title>
</head>
<body>
  <div id="output"></div>
  <script src="main-bundle.js"></script>
</body>
</html>
```

The `<div>` with the ID `"output"` is where the web app displays its output. `main-bundle.js` contains the bundled code.

9.8 `main.ts`

This is the TypeScript code of the web app:

```
import template from 'lodash/template';

const outputElement = document.getElementById('output');
if (outputElement) {
  const compiled = template(`
    <h1><%- heading %></h1>
    Current date and time: <%- dateTimeString %>
  `.trim());
  outputElement.innerHTML = compiled({
    heading: 'ts-demo-webpack',
    dateTimeString: new Date().toISOString(),
  });
}
```

- Step 1: We use Lodash's function `template()` to turn a string with custom template syntax into a function `compiled()` that maps data to HTML. The string defines two blanks to be filled in via data:
  - `<%- heading %>`
  - `<%- dateTimeString %>`
- Step 2: Apply `compiled()` to the data (an object with two properties) to generate HTML.

9.9 Installing, building and running the web app

First we need to install all npm packages that our web app depends on:

```
npm install
```

Then we need to run webpack (which was installed during the previous step) via a script in `package.json`:

```
npm run wpw
```

From now on, webpack watches the files in the repository for changes and rebuilds the web app whenever it detects any.

In a different command line, we can now start a web server that serves the contents of `build/` on localhost:

```
npm run serve
```

If we go to the URL printed out by the web server, we can see the web app in action.

Note that simple reloading may not be enough to see the results after changes – due to caching. You may have to force-reload by pressing shift when reloading.

9.9.1 Building in Visual Studio Code

Instead of building from a command line, we can also do that from within Visual Studio Code, via a so-called *build task*:

- Execute "Configure Default Build Task…" from the "Terminal" menu.
- Choose "npm: wpw".
- A *problem matcher* handles the conversion of tool output into lists of *problems* (infos, warning, and errors). The default works well in this case. If you want to be explicit, you can specify a value in `.vscode/tasks.json`:

  ```
  "problemMatcher": ["$tsc-watch"],
  ```

We can now start webpack via "Run Build Task…" from the "Terminal" menu.

9.10 Using webpack without a loader: `webpack-no-loader.config.js`

Instead of using on `ts-loader`, we can also first compile our TypeScript files to JavaScript files and then bundle those via webpack. How the first of those two steps works, is described in the previous chapter.

We now don't have to configure `ts-loader` and our webpack configuration file is simpler:

```
const path = require('path');

module.exports = {
  entry: {
    main: "./dist/src/main.js",
  },
  output: {
    path: path.join(__dirname, 'build'),
    filename: '[name]-bundle.js',
  },
  plugins: [
    new CopyWebpackPlugin([
      {
        from: './html',
      }
    ]),
  ],
};
```

Note that `entry.main` is different. In the other config file, it is:

```
"./ts/src/main.ts"
```

Why would we want to produce intermediate files before bundling them? One benefit is that we can use Node.js to run unit tests for some of the TypeScript code.

Chapter 10

Strategies for migrating to TypeScript

Contents

This chapter gives an overview of strategies for migrating code bases from JavaScript to TypeScript. It also mentions material for further reading.

10.1 Three strategies

These are three strategies for migrating to TypeScript:

- We can support a mix of JavaScript and TypeScript files for our code base. We start with only JavaScript files and then switch more and more files to TypeScript.
- We can keep our current (non-TypeScript) build process and our JavaScript-only code base. We add static type information via JSDoc comments and use TypeScript as a type checker (not as a compiler). Once everything is correctly typed, we switch to TypeScript for building.
- For large projects, there may be too many TypeScript errors during migration. Then snapshot tests can help us find fixed errors and new errors.

More information:

- "Migrating from JavaScript" in the TypeScript Handbook

10.2 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.

10.3 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:

- §4.4 "Using the TypeScript compiler for plain JavaScript files"
- "How we gradually migrated to TypeScript at Unsplash" by Oliver Joseph Ash

10.4 Strategy: migrating large projects by snapshot testing the TypeScript errors

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

10.5 Conclusion

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. Talk to your team w.r.t. prioritization:
  - Sometimes finishing the migration quickly may take priority.
  - Sometimes the code remaining fully functional during the migration may be more important.
  - And so on…

Part III

Digging deeper

Chapter 11

What is a type in TypeScript? Two perspectives

Contents

What are types in TypeScript? This chapter describes two perspectives that help with understanding them.

11.1 Three questions for each perspective

The following three 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 `myVariable` to have the type `MyType`?

   ```
   let myVariable: MyType = /*...*/;
   ```

2. Assignability: A `SourceType` is *assignable* to a `TargetType` if each storage location of type `SourceType` can be assigned to any storage location of type `TargetType` (line A).

   ```
   let source: SourceType = /*...*/;
   let target: TargetType = source; // (A)
   ```

3. How is `TypeUnion` derived from `Type1`, `Type2`, and `Type3`?

   ```
   type TypeUnion = Type1 | Type2 | Type3;
   ```

11.2 Perspective 1: types are sets of values

From this perspective, a type is a set of values:

1. If `myVariable` has the type `MyType`, that means that all values that can be assigned to `myVariable` must be elements of the set `MyType`.
2. `SourceType` is assignable to `TargetType` if `SourceType` is a subset of `TargetType`. As a consequence, all values allowed by `SourceType` are also allowed by `TargetType`.
3. The union type of the types `Type1`, `Type2`, and `Type3` is the set-theoretic union of the sets that define them.

11.3 Perspective 2: type compatibility relationships

From this perspective, we are not concerned with values and how they flow when code is executed. Instead, we take a more static view:

- 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.
- When a source location is connected to a target location via an assignment, a function call, etc., then the type of the source location must be compatible with the type of the target location. The TypeScript specification defines type compatibility via so-called *type relationships*.
- The type relationship *assignment compatibility* defines when a source type `S` is assignable to a target type `T`:
  - `S` and `T` are identical types.
  - `S` or `T` is the type `any`.
  - Etc.

Let's consider the questions:

1. The static type of a variable determines what can be assigned to it. That always depends on static types. For example, in a function call `toString(123)`, the static type of `123` must be assignable to the static type of the parameter of `toString()`.
2. `myVariable` has type `MyType` if the static type of `myVariable` is assignable to `MyType`.
3. `SourceType` is assignable to `TargetType` if they are assignment-compatible.
4. How union types work is defined via the type relationship *apparent members*.

An interesting trait of TypeScript's type system is that the same variable can have different static types at different locations:

```
// %inferred-type: any[]
const arr = [];

arr.push(123);
```

```
// %inferred-type: number[]
arr;

arr.push('abc');
// %inferred-type: (string | number)[]
arr;
```

11.4 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:

- 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)

This often means checking if `Src` is a subtype of `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"). One type is a subtype of another if their subtype relationship was defined explicitly.
  - Languages with nominal typing are C++, Java, C#, Swift, and Rust.
- In a *structural* type system, two static types are equal if they have the same structure (if their parts have the same names and the same types). One type `Sub` is a subtype of another type `Sup` if `Sub` has all parts of `Sup` (and possibly others) and each part of `Sub` has a subtype of the corresponding part of `Sup`.
  - Languages with structural typing are OCaml/ReasonML and TypeScript.

The following code produces a type error in line A with nominal type systems, but is legal with TypeScript's structural type system because class `A` and class `B` have the same structure:

```
class A {
  name = 'A';
}
class B {
  name = 'B';
}
const someVariable: A = new B(); // (A)
```

TypeScript's interfaces also work structurally – they don't have to be implemented in order to match:

```
interface Point {
  x: number;
  y: number;
}
const point: Point = {x: 1, y: 2}; // OK
```

11.5 Further reading

- Chapter “Type Compatibility” in the TypeScript Handbook
- Section “Type Relationships” in the TypeScript Specification

Chapter 12

TypeScript enums: How do they work? What can they be used for?

Contents

This chapter answers the following two questions:

- How do TypeScript's enums work?
- What can they be used for?

In the next chapter, we take a look at alternatives to enums.

12.1 The basics

`boolean` is a type with a finite amount of values: `false` and `true`. With enums, TypeScript lets us define similar types ourselves.

12.1.1 Numeric enums

This is a numeric enum:

```
enum NoYes {
  No = 0,
  Yes = 1, // trailing comma
}

assert.equal(NoYes.No, 0);
assert.equal(NoYes.Yes, 1);
```

Explanations:

- The entries `No` and `Yes` are called the *members* of the enum `NoYes`.
- Each enum member has a *name* and a *value*. For example, the first member has the name `No` and the value `0`.
- The part of a member definition that starts with an equals sign and specifies a value is called an *initializer*.
- As in object literals, trailing commas are allowed and ignored.

We can use members as if they were literals such as `true`, `123`, or `'abc'` – for example:

```
function toGerman(value: NoYes) {
  switch (value) {
    case NoYes.No:
      return 'Nein';
    case NoYes.Yes:
      return 'Ja';
  }
}
assert.equal(toGerman(NoYes.No), 'Nein');
assert.equal(toGerman(NoYes.Yes), 'Ja');
```

12.1.2 String-based enums

Instead of numbers, we can also use strings as enum member values:

```
enum NoYes {
  No = 'No',
  Yes = 'Yes',
}

assert.equal(NoYes.No, 'No');
assert.equal(NoYes.Yes, 'Yes');
```

12.1.3 Heterogeneous enums

The last kind of enums is called *heterogeneous*. The member values of a heterogeneous enum are a mix of numbers and strings:

```
enum Enum {
  One = 'One',
  Two = 'Two',
  Three = 3,
  Four = 4,
}
assert.deepEqual(
  [Enum.One, Enum.Two, Enum.Three, Enum.Four],
  ['One', 'Two', 3, 4]
);
```

Heterogeneous enums are not used often because they have few applications.

Alas, TypeScript only supports numbers and strings as enum member values. Other values, such as symbols, are not allowed.

12.1.4 Omitting initializers

We can omit initializers in two cases:

- We can omit the initializer of the first member. Then that member has the value 0 (zero).
- We can omit the initializer of a member if the previous member has a number value. Then the current member has that value plus one.

This is a numeric enum without any initializers:

```
enum NoYes {
  No,
  Yes,
}
assert.equal(NoYes.No, 0);
assert.equal(NoYes.Yes, 1);
```

This is a heterogeneous enum where some initializers are omitted:

```
enum Enum {
  A,
  B,
  C = 'C',
  D = 'D',
  E = 8, // (A)
  F,
}
assert.deepEqual(
  [Enum.A, Enum.B, Enum.C, Enum.D, Enum.E, Enum.F],
  [0, 1, 'C', 'D', 8, 9]
);
```

Note that we can't omit the initializer in line A because the value of the preceding member is not a number.

12.1.5 Casing of enum member names

There are several precedents for naming constants (in enums or elsewhere):

- Traditionally, JavaScript has used all-caps names, which is a convention it inherited from Java and C:
  - `Number.MAX_VALUE`
  - `Math.SQRT2`
- Well-known symbols are are camel-cased and start with lowercase letters because they are related to property names:
  - `Symbol.asyncIterator`
- The TypeScript manual uses camel-cased names that start with uppercase letters. This is the standard TypeScript style and we used it for the `NoYes` enum.

12.1.6 Quoting enum member names

Similar to JavaScript objects, we can quote the names of enum members:

```
enum HttpRequestField {
  'Accept',
  'Accept-Charset',
  'Accept-Datetime',
  'Accept-Encoding',
  'Accept-Language',
}
assert.equal(HttpRequestField['Accept-Charset'], 1);
```

There is no way to compute the names of enum members. Object literals support computed property keys via square brackets.

12.2 Specifying enum member values (advanced)

TypeScript distinguishes three kinds of enum members, by how they are initialized:

- A *literal enum member*:
  - either has no initializer
  - or is initialized via a number literal or a string literal.
- A *constant enum member* is initialized via an expression whose result can be computed at compile time.
- A *computed enum member* is initialized via an arbitrary expression.

So far, we have only used literal members.

In the previous list, members that are mentioned earlier are less flexible but support more features. Read on for more information.

12.2.1 Literal enum members

An enum member is *literal* if its value is specified:

- either implicitly
- or via a number literal (incl. negated number literals)
- or via a string literal.

If an enum has only literal members, we can use those members as types (similar to how, e.g., number literals can be used as types):

```
enum NoYes {
  No = 'No',
  Yes = 'Yes',
}
function func(x: NoYes.No) { // (A)
  return x;
}

func(NoYes.No); // OK

// @ts-expect-error: Argument of type '"No"' is not assignable to
// parameter of type 'NoYes.No'.
func('No');

// @ts-expect-error: Argument of type 'NoYes.Yes' is not assignable to
// parameter of type 'NoYes.No'.
func(NoYes.Yes);
```

`NoYes.No` in line A is an *enum member type*.

Additionally, literal enums support exhaustiveness checks (which we'll look at later).

12.2.2 Constant enum members

An enum member is constant if its value can be computed at compile time. Therefore, we can either specify its value implicitly (that is, we let TypeScript specify it for us). Or we can specify it explicitly and are only allowed to use the following syntax:

- Number literals or string literals
- A reference to a previously defined constant enum member (in the current enum or in a previous enum)
- Parentheses
- The unary operators +, -, ~
- The binary operators +, -, *, /, %, <<, >>, >>>, &, |, ^

This is an example of an enum whose members are all constant (we'll see later how that enum is used):

```
enum Perm {
  UserRead     = 1 << 8, // bit 8
  UserWrite    = 1 << 7,
  UserExecute  = 1 << 6,
  GroupRead    = 1 << 5,
  GroupWrite   = 1 << 4,
  GroupExecute = 1 << 3,
  AllRead      = 1 << 2,
  AllWrite     = 1 << 1,
  AllExecute   = 1 << 0,
}
```

In general, constant members can't be used as types. However, exhaustiveness checks are still performed.

12.2.3 Computed enum members

The values of *computed enum members* can be specified via arbitrary expressions. For example:

```
enum NoYesNum {
  No = 123,
  Yes = Math.random(), // OK
}
```

This was a numeric enum. String-based enums and heterogeneous enums are more limited. For example, we cannot use method invocations to specify member values:

```
enum NoYesStr {
  No = 'No',
  // @ts-expect-error: Computed values are not permitted in
  // an enum with string valued members.
  Yes = ['Y', 'e', 's'].join(''),
}
```

TypeScript does not do exhaustiveness checks for computed enum members.

12.3 Downsides of numeric enums

12.3.1 Downside: logging

When logging members of numeric enums, we only see numbers:

```
enum NoYes { No, Yes }

console.log(NoYes.No);
console.log(NoYes.Yes);

// Output:
// 0
// 1
```

12.3.2 Downside: loose type-checking

When using the enum as a type, the values that are allowed statically are not just those of the enum members – any number is accepted:

```
enum NoYes { No, Yes }
function func(noYes: NoYes) {}
func(33); // no error!
```

Why aren't there stricter static checks? Daniel Rosenwasser explains:

> The behavior is motivated by bitwise operations. There are times when `SomeFlag.Foo | SomeFlag.Bar` is intended to produce another `SomeFlag`. Instead you end up with `number`, and you don't want to have to cast back to `SomeFlag`.
>
> I think if we did TypeScript over again and still had enums, we'd have made a separate construct for bit flags.

How enums are used for bit patterns is demonstrated soon in more detail.

12.3.3 Recommendation: prefer string-based enums

My recommendation is to prefer string-based enums (for brevity's sake, this chapter doesn't always follow this recommendation):

```
enum NoYes { No='No', Yes='Yes' }
```

On one hand, logging output is more useful for humans:

```
console.log(NoYes.No);
console.log(NoYes.Yes);

// Output:
// 'No'
// 'Yes'
```

On the other hand, we get stricter type checking:

```
function func(noYes: NoYes) {}

// @ts-expect-error: Argument of type '"abc"' is not assignable
// to parameter of type 'NoYes'.
func('abc');

// @ts-expect-error: Argument of type '"Yes"' is not assignable
// to parameter of type 'NoYes'.
func('Yes'); // (A)
```

Not even strings that are equal to values of members are allowed (line A).

12.4 Use cases for enums

12.4.1 Use case: bit patterns

In the Node.js file system module, several functions have the parameter `mode`. It specifies file permissions, via a numeric encoding that is a holdover from Unix:

- Permissions are specified for three categories of users:
  - User: the owner of the file
  - Group: the members of the group associated with the file
  - All: everyone
- Per category, the following permissions can be granted:
  - r (read): the users in the category are allowed to read the file
  - w (write): the users in the category are allowed to change the file
  - x (execute): the users in the category are allowed to run the file

That means that permissions can be represented by 9 bits (3 categories with 3 permissions each):

	User	Group	All
Permissions	r, w, x	r, w, x	r, w, x
Bit	8, 7, 6	5, 4, 3	2, 1, 0

Node.js doesn't do this, but we could use an enum to work with these flags:

```
enum Perm {
  UserRead     = 1 << 8, // bit 8
  UserWrite    = 1 << 7,
  UserExecute  = 1 << 6,
  GroupRead    = 1 << 5,
  GroupWrite   = 1 << 4,
  GroupExecute = 1 << 3,
  AllRead      = 1 << 2,
  AllWrite     = 1 << 1,
  AllExecute   = 1 << 0,
}
```

Bit patterns are combined via bitwise Or:

```
// User can change, read and execute.
// Everyone else can only read and execute.
assert.equal(
  Perm.UserRead | Perm.UserWrite | Perm.UserExecute |
  Perm.GroupRead | Perm.GroupExecute |
  Perm.AllRead | Perm.AllExecute,
  0o755);

// User can read and write.
// Group members can read.
// Everyone can’t access at all.
assert.equal(
  Perm.UserRead | Perm.UserWrite | Perm.GroupRead,
  0o640);
```

12.4.1.1 An alternative to bit patterns

The main idea behind bit patterns is that there is a set of flags and that any subset of those flags can be chosen.

Therefore, using real sets to choose subsets is a more straightforward way of performing the same task:

```
enum Perm {
  UserRead = 'UserRead',
  UserWrite = 'UserWrite',
  UserExecute = 'UserExecute',
  GroupRead = 'GroupRead',
  GroupWrite = 'GroupWrite',
  GroupExecute = 'GroupExecute',
  AllRead = 'AllRead',
  AllWrite = 'AllWrite',
  AllExecute = 'AllExecute',
}
function writeFileSync(
  thePath: string, permissions: Set<Perm>, content: string) {
  // ···
}
writeFileSync(
  '/tmp/hello.txt',
  new Set([Perm.UserRead, Perm.UserWrite, Perm.GroupRead]),
  'Hello!');
```

12.4.2 Use case: multiple constants

Sometimes, we have sets of constants that belong together:

```
const off = Symbol('off');
const info = Symbol('info');
const warn = Symbol('warn');
const error = Symbol('error');
```

This is a good use case for an enum:

```
enum LogLevel {
  off = 'off',
  info = 'info',
  warn = 'warn',
  error = 'error',
}
```

One benefit of the enum is that the constant names are grouped and nested inside the namespace `LogLevel`.

Another one is that we automatically get the type `LogLevel` for them. If we want such a type for the constants, we need more work:

```
type LogLevel =
  | typeof off
  | typeof info
  | typeof warn
  | typeof error
;
```

For more information on this approach, see §13.1.3 "Unions of symbol singleton types".

12.4.3 Use case: more self-descriptive than booleans

When booleans are used to represent alternatives, enums are usually more self-descriptive.

12.4.3.1 Boolean-ish example: ordered vs. unordered lists

For example, to represent whether a list is ordered or not, we can use a boolean:

```
class List1 {
  isOrdered: boolean;
  // ···
}
```

However, an enum is more self-descriptive and has the additional benefit that we can add more alternatives later if we need to.

```
enum ListKind { ordered, unordered }
class List2 {
  listKind: ListKind;
  // ···
}
```

12.4.3.2 Boolean-ish example: error handling modes

Similarly, we can specify how to handle errors via a boolean value:

```
function convertToHtml1(markdown: string, throwOnError: boolean) {
  // ···
}
```

Or we can do so via an enum value:

```
enum ErrorHandling {
  throwOnError = 'throwOnError',
  showErrorsInContent = 'showErrorsInContent',
}
function convertToHtml2(markdown: string, errorHandling: ErrorHandling) {
  // ···
}
```

12.4.4 Use case: better string constants

Consider the following function that creates regular expressions.

```
const GLOBAL = 'g';
const NOT_GLOBAL = '';
type Globalness = typeof GLOBAL | typeof NOT_GLOBAL;

function createRegExp(source: string,
  globalness: Globalness = NOT_GLOBAL) {
    return new RegExp(source, 'u' + globalness);
  }

assert.deepEqual(
  createRegExp('abc', GLOBAL),
  /abc/ug);

assert.deepEqual(
  createRegExp('abc', 'g'), // OK
  /abc/ug);
```

Instead of the string constants, we can use an enum:

```
enum Globalness {
  Global = 'g',
  notGlobal = '',
}

function createRegExp(source: string, globalness = Globalness.notGlobal) {
  return new RegExp(source, 'u' + globalness);
}

assert.deepEqual(
  createRegExp('abc', Globalness.Global),
  /abc/ug);

assert.deepEqual(
  // @ts-expect-error: Argument of type '"g"' is not assignable to parameter of type 'Glo
  createRegExp('abc', 'g'), // error
  /abc/ug);
```

What are the benefits of this approach?

- It is more concise.
- It is slightly safer: The type `Globalness` only accepts member names, not strings.

12.5 Enums at runtime

TypeScript compiles enums to JavaScript objects. As an example, take the following enum:

```
enum NoYes {
  No,
  Yes,
}
```

TypeScript compiles this enum to:

```
var NoYes;
(function (NoYes) {
  NoYes[NoYes["No"] = 0] = "No";
  NoYes[NoYes["Yes"] = 1] = "Yes";
})(NoYes || (NoYes = {}));
```

In this code, the following assignments are made:

```
NoYes["No"] = 0;
NoYes["Yes"] = 1;

NoYes[0] = "No";
NoYes[1] = "Yes";
```

There are two groups of assignments:

- The first two assignments map enum member names to values.
- The second two assignments map values to names. That enables *reverse mappings*, which we will look at next.

12.5.1 Reverse mappings

Given a numeric enum:

```
enum NoYes {
  No,
  Yes,
}
```

The normal mapping is from member names to member values:

```
// Static (= fixed) lookup:
assert.equal(NoYes.Yes, 1);

// Dynamic lookup:
assert.equal(NoYes['Yes'], 1);
```

Numeric enums also support a *reverse mapping* from member values to member names:

```
assert.equal(NoYes[1], 'Yes');
```

One use case for reverse mappings is printing the name of an enum member:

```
function getQualifiedName(value: NoYes) {
  return 'NoYes.' + NoYes[value];
}
assert.equal(
  getQualifiedName(NoYes.Yes), 'NoYes.Yes');
```

12.5.2 String-based enums at runtime

String-based enums have a simpler representation at runtime.

Consider the following enum.

```
enum NoYes {
  No = 'NO!',
  Yes = 'YES!',
}
```

It is compiled to this JavaScript code:

```
var NoYes;
(function (NoYes) {
    NoYes["No"] = "NO!";
    NoYes["Yes"] = "YES!";
})(NoYes || (NoYes = {}));
```

TypeScript does not support reverse mappings for string-based enums.

12.6 `const` enums

If an enum is prefixed with the keyword `const`, it doesn't have a representation at runtime. Instead, the values of its member are used directly.

12.6.1 Compiling non-const enums

To observe this effect, let us first examine the following non-const enum:

```
enum NoYes {
  No = 'No',
  Yes = 'Yes',
}

function toGerman(value: NoYes) {
  switch (value) {
    case NoYes.No:
      return 'Nein';
    case NoYes.Yes:
      return 'Ja';
  }
}
```

TypeScript compiles this code to:

```
"use strict";
var NoYes;
(function (NoYes) {
  NoYes["No"] = "No";
  NoYes["Yes"] = "Yes";
})(NoYes || (NoYes = {}));

function toGerman(value) {
  switch (value) {
    case NoYes.No:
      return 'Nein';
    case NoYes.Yes:
      return 'Ja';
  }
}
```

12.6.2 Compiling const enums

This is the same code as previously, but now the enum is const:

```
const enum NoYes {
  No,
  Yes,
}
function toGerman(value: NoYes) {
  switch (value) {
    case NoYes.No:
      return 'Nein';
    case NoYes.Yes:
      return 'Ja';
  }
}
```

Now the representation of the enum as a construct disappears and only the values of its members remain:

```
function toGerman(value) {
  switch (value) {
    case "No" /* No */:
      return 'Nein';
    case "Yes" /* Yes */:
      return 'Ja';
  }
}
```

12.7 Enums at compile time

12.7.1 Enums are objects

TypeScript treats (non-const) enums as if they were objects:

```
enum NoYes {
  No = 'No',
  Yes = 'Yes',
}
function func(obj: { No: string }) {
  return obj.No;
}
assert.equal(
  func(NoYes), // allowed statically!
  'No');
```

12.7.2 Safety checks for literal enums

When we accept an enum member value, we often want to make sure that:

- We don't receive illegal values.
- We don't forget to consider any enum member values. This is especially relevant if we add members later.

Read on for more information. We will be working with the following enum:

```
enum NoYes {
  No = 'No',
  Yes = 'Yes',
}
```

12.7.2.1 Protecting against illegal values

In the following code, we take two measures against illegal values:

```
function toGerman1(value: NoYes) {
  switch (value) {
    case NoYes.No:
      return 'Nein';
    case NoYes.Yes:
      return 'Ja';
    default:
      throw new TypeError('Unsupported value: ' + JSON.stringify(value));
  }
}

assert.throws(
  // @ts-expect-error: Argument of type '"Maybe"' is not assignable to
  // parameter of type 'NoYes'.
  () => toGerman1('Maybe'),
  /^TypeError: Unsupported value: "Maybe"$/);
```

The measures are:

- At compile time, the type `NoYes` prevents illegal values being passed to the parameter `value`.
- At runtime, the `default` case is used to throw an exception if there is an unexpected value.

12.7.2.2 Protecting against forgetting cases via exhaustiveness checks

We can take one more measure. The following code performs an *exhaustiveness check*: TypeScript will warn us if we forget to consider all enum members.

```
class UnsupportedValueError extends Error {
  constructor(value: never) {
    super('Unsupported value: ' + value);
  }
}

function toGerman2(value: NoYes) {
  switch (value) {
    case NoYes.No:
      return 'Nein';
    case NoYes.Yes:
      return 'Ja';
    default:
      throw new UnsupportedValueError(value);
  }
}
```

How does the exhaustiveness check work? For every case, TypeScript infers the type of `value`:

```
function toGerman2b(value: NoYes) {
  switch (value) {
    case NoYes.No:
      // %inferred-type: NoYes.No
      value;
      return 'Nein';
    case NoYes.Yes:
      // %inferred-type: NoYes.Yes
      value;
      return 'Ja';
    default:
      // %inferred-type: never
      value;
      throw new UnsupportedValueError(value);
  }
}
```

In the default case, TypeScript infers the type `never` for `value` because we never get there. If however, we add a member `.Maybe` to `NoYes`, then the inferred type of `value` is `NoYes.Maybe`. And that type is statically incompatible with the type `never` of the parameter of `new UnsupportedValueError()`. That's why we get the following error message at compile time:

```
Argument of type 'NoYes.Maybe' is not assignable to parameter of type 'never'.
```

Conveniently, this kind of exhaustiveness check also works with `if` statements:

```
function toGerman3(value: NoYes) {
  if (value === NoYes.No) {
    return 'Nein';
  } else if (value === NoYes.Yes) {
    return 'Ja';
  } else {
    throw new UnsupportedValueError(value);
  }
}
```

12.7.2.3 An alternative way of checking exhaustiveness

Alternatively, we also get an exhaustiveness check if we specify a return type:

```
function toGerman4(value: NoYes): string {
  switch (value) {
    case NoYes.No:
      const x: NoYes.No = value;
      return 'Nein';
    case NoYes.Yes:
      const y: NoYes.Yes = value;
      return 'Ja';
  }
}
```

If we add a member to `NoYes`, then TypeScript complains that `toGerman4()` may return `undefined`.

Downsides of this approach:

- This approach does not work with `if` statements (more information).
- No checks are performed at runtime.

12.7.3 keyof and enums

We can use the `keyof` type operator to create the type whose elements are the keys of the enum members. When we do so, we need to combine `keyof` with `typeof`:

```
enum HttpRequestKeyEnum {
  'Accept',
  'Accept-Charset',
  'Accept-Datetime',
  'Accept-Encoding',
  'Accept-Language',
}
// %inferred-type: "Accept" | "Accept-Charset" | "Accept-Datetime" |
// "Accept-Encoding" | "Accept-Language"
type HttpRequestKey = keyof typeof HttpRequestKeyEnum;

function getRequestHeaderValue(request: Request, key: HttpRequestKey) {
  // ···
}
```

12.7.3.1 Using keyof without typeof

If we use `keyof` without `typeof`, we get a different, less useful, type:

```
// %inferred-type: "toString" | "toFixed" | "toExponential" |
// "toPrecision" | "valueOf" | "toLocaleString"
type Keys = keyof HttpRequestKeyEnum;
```

`keyof HttpRequestKeyEnum` is the same as `keyof number`.

12.8 Acknowledgment

- Thanks to Disqus user `@spira_mirabilis` for their feedback to this chapter.

Chapter 13

Alternatives to enums in TypeScript

Contents

The previous chapter explored how TypeScript enums work. In this chapter, we take a look at alternatives to enums.

13.1 Unions of singleton values

An enum maps member names to member values. If we don't need or want the indirection, we can use a union of so-called *primitive literal types* – one per value. Before we can go into details, we need to learn about primitive literal types.

13.1.1 Primitive literal types

Quick recap: We can consider types to be sets of values.

A *singleton type* is a type with one element. Primitive literal types are singleton types:

```
type UndefinedLiteralType = undefined;
type NullLiteralType = null;

type BooleanLiteralType = true;
type NumericLiteralType = 123;
type BigIntLiteralType = 123n; // --target must be ES2020+
type StringLiteralType = 'abc';
```

`UndefinedLiteralType` is the type with the single element `undefined`, etc.

It is important to be aware of the two language levels at play here (we have already encountered those levels earlier in this book). Consider the following variable declaration:

```
const abc: 'abc' = 'abc';
```

- The first `'abc'` represents a type (a string literal type).
- The second `'abc'` represents a value.

Two use cases for primitive literal types are:

- Overloading on string parameters which enables the first argument of the following method call to determine the type of the second argument:

  ```
  elem.addEventListener('click', myEventHandler);
  ```

- We can use a union of primitive literal types to define a type by enumerating its members:

  ```
  type IceCreamFlavor = 'vanilla' | 'chocolate' | 'strawberry';
  ```

Read on for more information about the second use case.

13.1.2 Unions of string literal types

We'll start with an enum and convert it to a union of string literal types.

```
enum NoYesEnum {
  No = 'No',
  Yes = 'Yes',
}
function toGerman1(value: NoYesEnum): string {
  switch (value) {
    case NoYesEnum.No:
      return 'Nein';
    case NoYesEnum.Yes:
      return 'Ja';
  }
}
```

```
assert.equal(toGerman1(NoYesEnum.No), 'Nein');
assert.equal(toGerman1(NoYesEnum.Yes), 'Ja');
```

`NoYesStrings` is the union type version of `NoYesEnum`:

```
type NoYesStrings = 'No' | 'Yes';

function toGerman2(value: NoYesStrings): string {
  switch (value) {
    case 'No':
      return 'Nein';
    case 'Yes':
      return 'Ja';
  }
}
assert.equal(toGerman2('No'), 'Nein');
assert.equal(toGerman2('Yes'), 'Ja');
```

The type `NoYesStrings` is the union of the string literal types `'No'` and `'Yes'`. The union type operator `|` is related to the set-theoretic union operator `∪`.

13.1.2.1 Unions of string literal types can be checked for exhaustiveness

The following code demonstrates that exhaustiveness checks work for unions of string literal types:

```
// @ts-expect-error: Function lacks ending return statement and
// return type does not include 'undefined'. (2366)
function toGerman3(value: NoYesStrings): string {
  switch (value) {
    case 'Yes':
      return 'Ja';
  }
}
```

We forgot the case for `'No'` and TypeScript warns us that the function may return values that are not strings.

We could have also checked exhaustiveness more explicitly:

```
class UnsupportedValueError extends Error {
  constructor(value: never) {
    super('Unsupported value: ' + value);
  }
}

function toGerman4(value: NoYesStrings): string {
  switch (value) {
    case 'Yes':
      return 'Ja';
    default:
      // @ts-expect-error: Argument of type '"No"' is not
```

```
        // assignable to parameter of type 'never'. (2345)
        throw new UnsupportedValueError(value);
    }
  }
```

Now TypeScript warns us that we reach the `default` case if `value` is `'No'`.

More information on exhaustiveness checking

For more information on this topic, see §12.7.2.2 "Protecting against forgetting cases via exhaustiveness checks".

13.1.2.2 Downside: unions of string literals are less type-safe

One downside of string literal unions is that non-member values can mistaken for members:

```
type Spanish = 'no' | 'sí';
type English = 'no' | 'yes';

const spanishWord: Spanish = 'no';
const englishWord: English = spanishWord;
```

This is logical because the Spanish `'no'` and the English `'no'` are the same value. The actual problem is that there is no way to give them different identities.

13.1.3 Unions of symbol singleton types

13.1.3.1 Example: `LogLevel`

Instead of unions of string literal types, we can also use unions of symbol singleton types. Let's start with a different enum this time:

```
enum LogLevel {
  off = 'off',
  info = 'info',
  warn = 'warn',
  error = 'error',
}
```

Translated to a union of symbol singleton types, it looks as follows:

```
const off = Symbol('off');
const info = Symbol('info');
const warn = Symbol('warn');
const error = Symbol('error');

// %inferred-type: unique symbol | unique symbol |
// unique symbol | unique symbol
type LogLevel =
  | typeof off
```

```
    | typeof info
    | typeof warn
    | typeof error
  ;
```

Why do we need `typeof` here? `off` etc. are values and can't appear in type equations. The type operator `typeof` fixes this issue by converting values to types.

Let's consider two variations of the previous example.

13.1.3.2 Variation #1: inlined symbols

Can we inline the symbols (instead of referring to separate `const` declarations)? Alas, the operand of the type operator `typeof` must be an identifier or a "path" of identifiers separated by dots. Therefore, this syntax is illegal:

```
type LogLevel = typeof Symbol('off') | ···
```

13.1.3.3 Variation #2: `let` instead of `const`

Can we use `let` instead of `const` to declare the variables? (That's not necessarily an improvement but still an interesting question.)

We can't because we need the narrower types that TypeScript infers for `const`-declared variables:

```
// %inferred-type: unique symbol
const constSymbol = Symbol('constSymbol');

// %inferred-type: symbol
let letSymbol1 = Symbol('letSymbol1');
```

With `let`, `LogLevel` would simply have been an alias for `symbol`.

`const` assertions normally solve this kind of problem. But they don't work in this case:

```
// @ts-expect-error: A 'const' assertions can only be applied to references to enum
// members, or string, number, boolean, array, or object literals. (1355)
let letSymbol2 = Symbol('letSymbol2') as const;
```

13.1.3.4 Using `LogLevel` in a function

The following function translates members of `LogLevel` to strings:

```
function getName(logLevel: LogLevel): string {
  switch (logLevel) {
    case off:
      return 'off';
    case info:
      return 'info';
    case warn:
      return 'warn';
    case error:
      return 'error';
  }
}

assert.equal(
  getName(warn), 'warn');
```

13.1.3.5 Unions of symbol singleton types vs. unions of string literal types

How do the two approaches compare?

- Exhaustiveness checks work for both.
- Using symbols is more verbose.
- Each symbol "literal" creates a unique symbol that can't be confused with any other symbol. That's not true for string literals. Read on for details.

Recall this example where the Spanish `'no'` was confused with the English `'no'`:

```
type Spanish = 'no' | 'sí';
type English = 'no' | 'yes';

const spanishWord: Spanish = 'no';
const englishWord: English = spanishWord;
```

If we use symbols, we don't have this problem:

```
const spanishNo = Symbol('no');
const spanishSí = Symbol('sí');
type Spanish = typeof spanishNo | typeof spanishSí;

const englishNo = Symbol('no');
const englishYes = Symbol('yes');
type English = typeof englishNo | typeof englishYes;

const spanishWord: Spanish = spanishNo;
// @ts-expect-error: Type 'unique symbol' is not assignable to type 'English'. (2322)
const englishWord: English = spanishNo;
```

13.1.4 Conclusion of this section: union types vs. enums

Union types and enums have some things in common:

- We can auto-complete member values. However, we do it differently:
  - With enums, we get auto-completion after the enum name and a dot.
  - With union types, we have to explicitly trigger auto-completion.
- Exhaustiveness checks also work for both.

But they also differ. Downsides of unions of symbol singleton types are:

- They are slightly verbose.
- There is no namespace for their members.

- It's slightly harder to migrate from them to different constructs (should it be necessary): It's easier to find where enum member values are mentioned.

Upsides of unions of symbol singleton types are:

- They are not a custom TypeScript language construct and therefore closer to plain JavaScript.
- String enums are only type-safe at compile time. Unions of symbol singleton types are additionally type-safe at runtime.
  - This matters especially if our compiled TypeScript code interacts with plain JavaScript code.

13.2 Discriminated unions

Discriminated unions are related to algebraic data types in functional programming languages.

To understand how they work, consider the data structure *syntax tree* that represents expressions such as:

```
1 + 2 + 3
```

A syntax tree is either:

- A number
- The addition of two syntax trees

Next steps:

1. We'll start by creating an object-oriented class hierarchy for syntax trees.
2. Then we'll transform it into something slightly more functional.
3. And finally, we'll end up with a discriminated union.

13.2.1 Step 1: the syntax tree as a class hierarchy

This is a typical object-oriented implementation of a syntax tree:

```
// Abstract = can't be instantiated via `new`
abstract class SyntaxTree1 {}
class NumberValue1 extends SyntaxTree1 {
  constructor(public numberValue: number) {
    super();
  }
}
class Addition1 extends SyntaxTree1 {
  constructor(public operand1: SyntaxTree1, public operand2: SyntaxTree1) {
    super();
  }
}
```

`SyntaxTree1` is the superclass of `NumberValue1` and `Addition1`. The keyword `public` is syntactic sugar for:

- Declaring the instance property `.numberValue`
- Initializing this property via the parameter `numberValue`

This is an example of using `SyntaxTree1`:

```
const tree = new Addition1(
  new NumberValue1(1),
  new Addition1(
    new NumberValue1(2),
    new NumberValue1(3), // trailing comma
  ), // trailing comma
);
```

Note: Trailing commas in argument lists are allowed in JavaScript since ECMAScript 2016.

13.2.2 Step 2: the syntax tree as a union type of classes

If we define the syntax tree via a union type (line A), we don't need object-oriented inheritance:

```
class NumberValue2 {
  constructor(public numberValue: number) {}
}
class Addition2 {
  constructor(public operand1: SyntaxTree2, public operand2: SyntaxTree2) {}
}
type SyntaxTree2 = NumberValue2 | Addition2; // (A)
```

Since `NumberValue2` and `Addition2` don't have a superclass, they don't need to invoke `super()` in their constructors.

Interestingly, we create trees in the same manner as before:

```
const tree = new Addition2(
  new NumberValue2(1),
  new Addition2(
    new NumberValue2(2),
    new NumberValue2(3),
  ),
);
```

13.2.3 Step 3: the syntax tree as a discriminated union

Finally, we get to discriminated unions. These are the type definitions for `SyntaxTree3`:

```
interface NumberValue3 {
  kind: 'number-value';
  numberValue: number;
}
interface Addition3 {
  kind: 'addition';
```

```
    operand1: SyntaxTree3;
    operand2: SyntaxTree3;
  }
  type SyntaxTree3 = NumberValue3 | Addition3;
```

We have switched from classes to interfaces and therefore from instances of classes to plain objects.

The interfaces of a discriminated union must have at least one property in common and that property must have a different value for each one of them. That property is called the *discriminant* or *tag*. The discriminant of `SyntaxTree3` is `.kind`. Its types are string literal types.

Compare:

- The direct class of an instance is determined by its prototype.
- The type of a member of a discriminated union is determined by its discriminant.

This is an object that matches `SyntaxTree3`:

```
const tree: SyntaxTree3 = { // (A)
  kind: 'addition',
  operand1: {
    kind: 'number-value',
    numberValue: 1,
  },
  operand2: {
    kind: 'addition',
    operand1: {
      kind: 'number-value',
      numberValue: 2,
    },
    operand2: {
      kind: 'number-value',
      numberValue: 3,
    },
  }
};
```

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.

In the next example, the type of `tree` is a discriminated union. Every time we check its discriminant (line C), TypeScript updates its static type accordingly:

```
function getNumberValue(tree: SyntaxTree3) {
  // %inferred-type: SyntaxTree3
  tree; // (A)

  // @ts-expect-error: Property 'numberValue' does not exist on type 'SyntaxTree3'.
  // Property 'numberValue' does not exist on type 'Addition3'.(2339)
  tree.numberValue; // (B)
```

```
    if (tree.kind === 'number-value') { // (C)
      // %inferred-type: NumberValue3
      tree; // (D)
      return tree.numberValue; // OK!
    }
    return null;
  }
```

In line A, we haven't checked the discriminant `.kind`, yet. Therefore, the current type of `tree` is still `SyntaxTree3` and we can't access property `.numberValue` in line B (because only one of the types of the union has this property).

In line D, TypeScript knows that `.kind` is `'number-value'` and can therefore infer the type `NumberValue3` for `tree`. That's why accessing `.numberValue` in the next line is OK, this time.

13.2.3.1 Implementing functions for discriminated unions

We conclude this step with an example of how to implement functions for discriminated unions.

If there is an operation that can be applied to members of all subtypes, the approaches for classes and discriminated unions differ:

- Object-oriented approach: With classes, it is common to use a polymorphic method where each class has a different implementation.
- Functional approach: With discriminated unions, it is common to use a single function that handles all possibles cases and decides what to do by examining the discriminant of its parameter.

The following example demonstrates the functional approach. The discriminant is examined in line A and determines which of the two `switch` cases is executed.

```
function syntaxTreeToString(tree: SyntaxTree3): string {
  switch (tree.kind) { // (A)
    case 'addition':
      return syntaxTreeToString(tree.operand1)
        + ' + ' + syntaxTreeToString(tree.operand2);
    case 'number-value':
      return String(tree.numberValue);
  }
}

assert.equal(syntaxTreeToString(tree), '1 + 2 + 3');
```

Note that TypeScript performs exhaustiveness checking for discriminated unions: If we forget a case, TypeScript will warn us.

This is the object-oriented version of the previous code:

```
abstract class SyntaxTree1 {
  // Abstract = enforce that all subclasses implement this method:
```

```
  abstract toString(): string;
}
class NumberValue1 extends SyntaxTree1 {
  constructor(public numberValue: number) {
    super();
  }
  toString(): string {
    return String(this.numberValue);
  }
}
class Addition1 extends SyntaxTree1 {
  constructor(public operand1: SyntaxTree1, public operand2: SyntaxTree1) {
    super();
  }
  toString(): string {
    return this.operand1.toString() + ' + ' + this.operand2.toString();
  }
}

const tree = new Addition1(
  new NumberValue1(1),
  new Addition1(
    new NumberValue1(2),
    new NumberValue1(3),
  ),
);

assert.equal(tree.toString(), '1 + 2 + 3');
```

13.2.3.2 Extensibility: object-oriented approach vs. functional approach

Each approach does one kind of extensibility well:

- With the object-oriented approach, 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 the functional approach, we have to modify each function if we want to add a new type. In contrast, adding new operations is simple.

13.2.4 Discriminated unions vs. normal union types

Discriminated unions and normal union types have two things in common:

- There is no namespace for member values.
- TypeScript performs exhaustiveness checking.

The next two subsections explore two advantages of discriminated unions over normal unions:

13.2.4.1 Benefit: descriptive property names

With discriminated unions, values get descriptive property names. Let's compare:

Normal union:

```
type FileGenerator = (webPath: string) => string;
type FileSource1 = string|FileGenerator;
```

Discriminated union:

```
interface FileSourceFile {
  type: 'FileSourceFile',
  nativePath: string,
}
interface FileSourceGenerator {
  type: 'FileSourceGenerator',
  fileGenerator: FileGenerator,
}
type FileSource2 = FileSourceFile | FileSourceGenerator;
```

Now people who read the source code immediately know what the string is: a native pathname.

13.2.4.2 Benefit: We can also use it when the parts are indistinguishable

The following discriminated union cannot be implemented as a normal union because we can't distinguish the types of the union in TypeScript.

```
interface TemperatureCelsius {
  type: 'TemperatureCelsius',
  value: number,
}
interface TemperatureFahrenheit {
  type: 'TemperatureFahrenheit',
  value: number,
}
type Temperature = TemperatureCelsius | TemperatureFahrenheit;
```

13.3 Object literals as enums

The following pattern for implementing enums is common in JavaScript:

```
const Color = {
  red: Symbol('red'),
  green: Symbol('green'),
  blue: Symbol('blue'),
};
```

We can attempt to use it in TypeScript as follows:

```
// %inferred-type: symbol
Color.red; // (A)
```

```
// %inferred-type: symbol
type TColor2 = // (B)
  | typeof Color.red
  | typeof Color.green
  | typeof Color.blue
;

function toGerman(color: TColor): string {
  switch (color) {
    case Color.red:
      return 'rot';
    case Color.green:
      return 'grün';
    case Color.blue:
      return 'blau';
    default:
      // No exhaustiveness check (inferred type is not `never`):
      // %inferred-type: symbol
      color;

      // Prevent static error for return type:
      throw new Error();
  }
}
```

Alas, the type of each property of `Color` is `symbol` (line A) and `TColor` (line B) is an alias for `symbol`. As a consequence, we can pass any symbol to `toGerman()` and TypeScript won't complain at compile time:

```
assert.equal(
  toGerman(Color.green), 'grün');
assert.throws(
  () => toGerman(Symbol())); // no static error!
```

A `const` assertion often helps in this kind of situation but not this time:

```
const ConstColor = {
  red: Symbol('red'),
  green: Symbol('green'),
  blue: Symbol('blue'),
} as const;

// %inferred-type: symbol
ConstColor.red;
```

The only way to fix this is via constants:

```
const red = Symbol('red');
const green = Symbol('green');
const blue = Symbol('blue');
```

```
// %inferred-type: unique symbol
red;

// %inferred-type: unique symbol | unique symbol | unique symbol
type TColor2 = typeof red | typeof green | typeof blue;
```

13.3.1 Object literals with string-valued properties

```
const Color = {
  red: 'red',
  green: 'green',
  blue: 'blue',
} as const; // (A)

// %inferred-type: "red"
Color.red;

// %inferred-type: "red" | "green" | "blue"
type TColor =
  | typeof Color.red
  | typeof Color.green
  | typeof Color.blue
;
```

We need `as const` in line A so that the properties of `Color` don't have the more general type `string`. Then `TColor` also has a type that is more specific than `string`.

Compared to using an object with symbol-valued properties as an enum, string-valued properties are:

- Better at development time because we get exhaustiveness checks and can derive a narrow type for the values (without using external constants).
- Worse at runtime because strings can be mistaken for enum values.

13.3.2 Upsides and downsides of using object literals as enums

Upsides:

- We have a namespace for the values.
- We don't use a custom construct and are closer to plain JavaScript.
- We can derive a narrow type for enum values (if we use string-valued properties).
- Exhaustiveness checks are performed for such a type.

Downsides:

- No dynamic membership check is possible (without extra work).
- Non-enum values can be mistaken for enum values statically or at runtime (if we use string-valued properties).

13.4 Enum pattern

The following example demonstrates a Java-inspired enum pattern that works in plain JavaScript and TypeScript:

```
class Color {
  static red = new Color();
  static green = new Color();
  static blue = new Color();
}

// @ts-expect-error: Function lacks ending return statement and return type
// does not include 'undefined'. (2366)
function toGerman(color: Color): string { // (A)
  switch (color) {
    case Color.red:
      return 'rot';
    case Color.green:
      return 'grün';
    case Color.blue:
      return 'blau';
  }
}

assert.equal(toGerman(Color.blue), 'blau');
```

Alas, TypeScript doesn't perform exhaustiveness checks, which is why we get an error in line A.

13.5 Summary of enums and enum alternatives

The following table summarizes the characteristics of enums and their alternatives in TypeScript:

	Unique	Namesp.	Iter.	Mem. CT	Mem. RT	Exhaust.
Number enums	-	✔	✔	✔	-	✔
String enums	✔	✔	✔	✔	-	✔
String unions	-	-	-	✔	-	✔
Symbol unions	✔	-	-	✔	-	✔
Discrim. unions	- (1)	-	-	✔	- (2)	✔
Symbol properties	✔	✔	✔	-	-	-
String properties	-	✔	✔	✔	-	✔
Enum pattern	✔	✔	✔	✔	✔	-

Titles of table columns:

- Unique values: No non-enum value can be mistaken for an enum value.
- Namespace for enum keys

- Is it possible to iterate over enum values?
- Membership check for values at compile time: Is there a narrow type for the enum values?
- Membership check for values at runtime:
  - For the enum pattern, the runtime membership test is `instanceof`.
  - Note that a membership test can be implemented relatively easily if it is possible to iterate over enum values.
- Exhaustiveness check (statically by TypeScript)

Footnotes in table cells:

1. Discriminated unions are not really unique, but mistaking values for union members is relatively unlikely (especially if we use a unique name for the discriminant property).
2. If the discriminant property has a unique enough name, it can be used to check membership.

13.6 Acknowledgement

- Thanks to Kirill Sukhomlin for his suggestion on how to define `TColor` for an object literal.

Chapter 14

Adding special values to types

Contents

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.

14.1 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), as siblings of normal values.

As an example, consider the following interface for readable streams:

```
interface InputStream {
  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.

14.1.1 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 InputStream {
  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(is: InputStream) {
  let commentCount = 0;
  while (true) {
    const line = is.getNextLine();
    // @ts-expect-error: Object is possibly 'null'.(2531)
    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(is: InputStream) {
  let commentCount = 0;
  while (true) {
    const line = is.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`.

14.1.2 Adding a symbol to a type

We can also use values other than `null` as sentinels. Symbols and objects 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. For more information on the different language levels of values and types, see §7.7 "The two language levels: dynamic vs. static".

14.2 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 InputStream<T> {
  getNextValue(): T;
}
```

Whatever value we pick for `EOF`, there is a risk of someone creating an `InputStream<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).

14.2.1 Discriminated unions

A *discriminated union* is a union type over 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.

14.2.1.1 Example: `InputStreamValue`

In the following example, `InputStreamValue<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 InputStreamValue<T> = Eof | NormalValue<T>;
```

```
interface InputStream<T> {
  getNextValue(): InputStreamValue<T>;
}

function countValues<T>(is: InputStream<T>, data: T) {
  let valueCount = 0;
  while (true) {
    // %inferred-type: Eof | NormalValue<T>
    const value = is.getNextValue(); // (A)

    if (value.type === 'eof') break;

    // %inferred-type: NormalValue<T>
    value; // (B)

    if (value.data === data) { // (C)
      valueCount++;
    }
  }
  return valueCount;
}
```

Initially, the type of `value` is `InputStreamValue<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.

14.2.1.2 Example: `IteratorResult`

When deciding how to implement iterators, TC39 didn't want to use a fixed sentinel value. Otherwise, that value could appear in iterables and break code. 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>;
```

14.2.2 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
  }
}
```

Chapter 15

Typing objects

Contents

In this chapter, we will explore how objects and properties are typed statically in TypeScript.

15.1 Roles played by objects

In JavaScript, objects can play two roles (always at least one of them, sometimes mixtures):

- *Records* have a fixed amount of properties that are known at development time. Each property can have a different type.
- *Dictionaries* have an arbitrary number of properties whose names are not known at development time. All property keys (strings and/or symbols) have the same type, as have property values.

First and foremost, we will explore objects as records. We will briefly encounter objects as dictionaries later in this chapter.

15.2 Types for objects

There are two different general types for objects:

- `Object` with an uppercase "O" is the type of all instances of class `Object`:

  ```
  let obj1: Object;
  ```

- `object` with a lowercase "o" is the type of all non-primitive values:

  ```
  let obj2: object;
  ```

Objects can also be typed via their properties:

```
// Object type literal
let obj3: {prop: boolean};

// Interface
interface ObjectType {
  prop: boolean;
}
let obj4: ObjectType;
```

In the next sections, we'll examine all these ways of typing objects in more detail.

15.3 `Object` vs. `object` in TypeScript

15.3.1 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:

  ```
  > 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
```

15.3.2 `Object` (uppercase "O") in TypeScript: instances of class `Object`

Recall that each class `C` creates two entities:

- A constructor function `C`.
- An interface `C` that describes instances of the constructor function.

Similarly, TypeScript has two built-in interfaces:

- Interface `Object` specifies the properties of instances of `Object`, including the properties inherited from `Object.prototype`.
- Interface `ObjectConstructor` specifies the properties of class `Object`.

These are the interfaces:

```
interface Object { // (A)
  constructor: Function;
  toString(): string;
  toLocaleString(): string;
  valueOf(): Object;
  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; // (B)

    getPrototypeOf(o: any): any;

    // ···
  }
  declare var Object: ObjectConstructor; // (C)
```

Observations:

- We have both a variable whose name is `Object` (line C) and a type whose name is `Object` (line A).
- Direct instances of `Object` have no own properties, therefore `Object.prototype` also matches `Object` (line B).

15.3.3 `object` (lowercase "o") in TypeScript: non-primitive values

In TypeScript, `object` is the type of all non-primitive values (primitive values are `undefined`, `null`, booleans, numbers, bigints, strings). With this type, we can't access any properties of a value.

15.3.4 `Object` vs. `object`: primitive values

Interestingly, type `Object` also matches primitive values:

```
function func1(x: Object) { }
func1('abc'); // OK
```

Why is that? Primitive values have all the properties required by `Object` because they inherit `Object.prototype`:

```
> 'abc'.hasOwnProperty === Object.prototype.hasOwnProperty
true
```

Conversely, `object` does not match primitive values:

```
function func2(x: object) { }
// @ts-expect-error: Argument of type '"abc"' is not assignable to
// parameter of type 'object'. (2345)
func2('abc');
```

15.3.5 `Object` vs. `object`: incompatible property types

With type `Object`, TypeScript complains if an object has a property whose type conflicts with the corresponding property in interface `Object`:

```
// @ts-expect-error: Type '() => number' is not assignable to
// type '() => string'.
//   Type 'number' is not assignable to type 'string'. (2322)
const obj1: Object = { toString() { return 123 } };
```

With type `object`, TypeScript does not complain (because `object` does not specify any properties and there can't be any conflicts):

```
const obj2: object = { toString() { return 123 } };
```

15.4 Object type literals and interfaces

TypeScript has two ways of defining object types that are very similar:

```
// Object type literal
type ObjType1 = {
  a: boolean,
  b: number;
  c: string,
};

// Interface
interface ObjType2 {
  a: boolean,
  b: number;
  c: string,
}
```

We can use either semicolons or commas as separators. Trailing separators are allowed and optional.

15.4.1 Differences between object type literals and interfaces

In this section, we take a look at the most important differences between object type literals and interfaces.

15.4.1.1 Inlining

Object type literals can be inlined, while interfaces can't be:

```
// Inlined object type literal:
function f1(x: {prop: number}) {}

// Referenced interface:
function f2(x: ObjectInterface) {}
interface ObjectInterface {
  prop: number;
}
```

15.4.1.2 Duplicate names

Type aliases with duplicate names are illegal:

```
// @ts-expect-error: Duplicate identifier 'PersonAlias'. (2300)
type PersonAlias = {first: string};
```

```
// @ts-expect-error: Duplicate identifier 'PersonAlias'. (2300)
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',
};
```

15.4.1.3 Mapped types

For Mapped types (line A), we need to use object type literals:

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

type PointCopy1 = {
  [Key in keyof Point]: Point[Key]; // (A)
};

// Syntax error:
// interface PointCopy2 {
//   [Key in keyof Point]: Point[Key];
// };
```

More information on mapped types

Mapped types are beyond the current scope of this book. For more information, see the TypeScript Handbook.

15.4.1.4 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.result += str;
    return this;
  }
}
```

Source of this section

- GitHub issue "TypeScript: types vs. interfaces" by Johannes Ewald

From now on, "interface" means "interface or object type literal" (unless stated otherwise).

15.4.2 Interfaces work structurally in TypeScript

Interfaces work structurally – they don't have to be implemented in order to match:

```
interface Point {
  x: number;
  y: number;
}
const point: Point = {x: 1, y: 2}; // OK
```

For more information on this topic, see §11.4 "Nominal type systems vs. structural type systems".

15.4.3 Members of interfaces and object type literals

The constructs inside the bodies of interfaces and object type literals are called their *members*. These are the most common members:

```
interface ExampleInterface {
  // 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): ExampleInstance;
}
interface ExampleInstance {}
```

Let’s look at these members in more detail:

- Property signatures define properties:

  ```
  myProperty: boolean;
  ```

- Method signatures define methods:

  ```
  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.

  ```
  [key: string]: any;
  ```

  Note: The name `key` is only there for documentation purposes.

- Call signatures enable interfaces to describe functions:

  ```
  (num: number): string;
  ```

- Construct signatures enable interfaces to describe classes and constructor functions:

  ```
  new(str: string): ExampleInstance;
  ```

Property signatures should be self-explanatory. Call signatures and construct signatures are described later in this book. We’ll take a closer look at method signatures and index signatures next.

15.4.4 Method signatures

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

```
interface HasMethodDef {
  simpleMethod(flag: boolean): void;
}
interface HasFuncProp {
  simpleMethod: (flag: boolean) => void;
}

const objWithMethod: HasMethodDef = {
  simpleMethod(flag: boolean): void {},
};
const objWithMethod2: HasFuncProp = objWithMethod;

const objWithOrdinaryFunction: HasMethodDef = {
  simpleMethod: function (flag: boolean): void {},
};
const objWithOrdinaryFunction2: HasFuncProp = objWithOrdinaryFunction;
```

```
const objWithArrowFunction: HasMethodDef = {
  simpleMethod: (flag: boolean): void => {},
};
const objWithArrowFunction2: HasFuncProp = objWithArrowFunction;
```

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

15.4.5 Index signatures: objects as dicts

So far, we have only used interfaces for objects-as-records with fixed keys. 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 {
  return dict[english];
}
```

We use an index signature (line A) to express that `TranslationDict` is for objects that map string keys to string values:

```
interface TranslationDict {
  [key:string]: string; // (A)
}
const dict = {
  'yes': 'sí',
  'no': 'no',
  'maybe': 'tal vez',
};
assert.equal(
  translate(dict, 'maybe'),
  'tal vez');
```

15.4.5.1 Typing index signature keys

Index signature keys must be either `string` or `number`:

- Symbols are not allowed.
- `any` is not allowed.
- Union types (e.g. `string|number`) are not allowed. However, multiple index signatures can be used per interface.

15.4.5.2 String keys vs. number keys

Just like in plain JavaScript, TypeScript's number property keys are a subset of the string property keys (see "JavaScript for impatient programmers"). 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`:

```
interface StringAndNumberKeys {
  [key: string]: Object;
```

```
    [key: number]: RegExp;
  }

  // %inferred-type: (x: StringAndNumberKeys) =>
  // { str: Object; num: RegExp; }
  function f(x: StringAndNumberKeys) {
    return { str: x['abc'], num: x[123] };
  }
```

15.4.5.3 Index signatures vs. property signatures and method signatures

If there are both an index signature and property and/or method signatures in an interface, then the type of the index property value must also be a supertype of the type of the property value and/or method.

```
interface I1 {
  [key: string]: boolean;

  // @ts-expect-error: Property 'myProp' of type 'number' is not assignable
  // to string index type 'boolean'. (2411)
  myProp: number;

  // @ts-expect-error: Property 'myMethod' of type '() => string' is not
  // assignable to string index type 'boolean'. (2411)
  myMethod(): string;
}
```

In contrast, the following two interfaces produce no errors:

```
interface I2 {
  [key: string]: number;
  myProp: number;
}

interface I3 {
  [key: string]: () => string;
  myMethod(): string;
}
```

15.4.6 Interfaces describe instances of `Object`

All interfaces 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;
}
```

15.4.7 Excess property checks: When are extra properties allowed?

As an example, consider the following interface:

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

There are two ways (among others) in which this interface 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: Argument of type '{ x: number; y: number; z: number; }'
// is not assignable to parameter of type 'Point'.
//   Object literal may only specify known properties, and 'z' does not
//   exist in type 'Point'. (2345)
computeDistance({ x: 1, y: 2, z: 3 }); // error

computeDistance({x: 1, y: 2}); // OK
```

15.4.7.1 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 interface to demonstrate what that means.

```
interface Person {
  first: string;
  middle?: string;
  last: string;
}
function computeFullName(person: Person) { /*...*/ }
```

Property `.middle` is optional and can be omitted (optional properties are covered later in this chapter). 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: Argument of type '{ first: string; mdidle: string;
// last: string; }' is not assignable to parameter of type 'Person'.
//   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'});
```

15.4.7.2 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:

```
interface HasYear {
  year: number;
}

function getAge(obj: HasYear) {
  const yearNow = new Date().getFullYear();
  return yearNow - obj.year;
}
```

Without allowing excess properties for most values that are passed to `getAge()`, the usefulness of this function would be quite limited.

15.4.7.3 Empty interfaces allow excess properties

If an interface is empty (or the object type literal `{}` is used), excess properties are always allowed:

```
interface Empty { }
interface OneProp {
  myProp: number;
}

// @ts-expect-error: Type '{ myProp: number; anotherProp: number; }' is not
// assignable to type 'OneProp'.
//   Object literal may only specify known properties, and
//   'anotherProp' does not exist in type 'OneProp'. (2322)
const a: OneProp = { myProp: 1, anotherProp: 2 };
const b: Empty = {myProp: 1, anotherProp: 2}; // OK
```

15.4.7.4 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):

```
interface WithoutProperties {
  [key: string]: never;
}

// @ts-expect-error: Type 'number' is not assignable to type 'never'. (2322)
const a: WithoutProperties = { prop: 1 };
const b: WithoutProperties = {}; // OK
```

15.4.7.5 Allowing excess properties in object literals

What if we want to allow excess properties in object literals? As an example, consider interface `Point` and function `computeDistance1()`:

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

function computeDistance1(point: Point) { /*...*/ }

// @ts-expect-error: Argument of type '{ x: number; y: number; z: number; }'
// is not assignable to parameter of type 'Point'.
//   Object literal may only specify known properties, and 'z' does not
//   exist in type 'Point'. (2345)
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 interface `Point` so that it allows excess properties:

```
interface PointEtc extends Point {
  [key: string]: any;
}
function computeDistance3(point: PointEtc) { /*...*/ }

computeDistance3({ x: 1, y: 2, z: 3 }); // OK
```

We'll continue with two examples where TypeScript not allowing excess properties, is an issue.

15.4.7.5.1 Allowing excess properties: example `Incrementor`

In this example, we'd like to implement an `Incrementor`, but TypeScript doesn't allow the extra property `.counter`:

```
interface Incrementor {
  inc(): void
}
function createIncrementor(start = 0): Incrementor {
  return {
    // @ts-expect-error: Type '{ counter: number; inc(): void; }' is not
    // assignable to type 'Incrementor'.
    //   Object literal may only specify known properties, and
    //   'counter' does not exist in type 'Incrementor'. (2322)
    counter: start,
    inc() {
      // @ts-expect-error: Property 'counter' does not exist on type
      // 'Incrementor'. (2339)
      this.counter++;
    },
  };
}
```

Alas, even with a type assertion, there is still one type error:

```
function createIncrementor2(start = 0): Incrementor {
  return {
    counter: start,
    inc() {
      // @ts-expect-error: Property 'counter' does not exist on type
      // 'Incrementor'. (2339)
      this.counter++;
    },
  } as Incrementor;
}
```

We can either add an index signature to interface `Incrementor`. Or – especially if that is not possible – we can introduce an intermediate variable:

```
function createIncrementor3(start = 0): Incrementor {
  const incrementor = {
    counter: start,
    inc() {
      this.counter++;
    },
  };
  return incrementor;
}
```

15.4.7.5.2 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.

15.5 Type inference

These are the types that TypeScript infers for objects that are created via various means:

```
// %inferred-type: Object
const obj1 = new Object();

// %inferred-type: any
const obj2 = Object.create(null);

// %inferred-type: {}
const obj3 = {};

// %inferred-type: { prop: number; }
const obj4 = {prop: 123};

// %inferred-type: object
const obj5 = Reflect.getPrototypeOf({});
```

In principle, the return type of `Object.create()` could be `object`. However, `any` allows us to add and change properties of the result.

15.6 Other features of interfaces

15.6.1 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:

```
interface 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'};
```

15.6.1.1 Optional vs. `undefined|string`

What is the difference between `.prop1` and `.prop2`?

```
interface Interf {
  prop1?: string;
  prop2: undefined | string;
}
```

An optional property can do everything that `undefined|string` can. We can even use the value `undefined` for the former:

```
const obj1: Interf = { prop1: undefined, prop2: undefined };
```

However, only `.prop1` can be omitted:

```
const obj2: Interf = { prop2: undefined };

// @ts-expect-error: Property 'prop2' is missing in type '{}' but required
// in type 'Interf'. (2741)
const obj3: Interf = { };
```

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.

15.6.2 Read-only properties

In the following example, property `.prop` is read-only:

```
interface MyInterface {
  readonly prop: number;
}
```

As a consequence, we can read it, but we can't change it:

```
const obj: MyInterface = {
  prop: 1,
};

console.log(obj.prop); // OK

// @ts-expect-error: Cannot assign to 'prop' because it is a read-only
// property. (2540)
obj.prop = 2;
```

15.7 JavaScript's prototype chains and TypeScript's types

TypeScript doesn't distinguish own and inherited properties. They are all simply considered to be properties.

```
interface MyInterface {
  toString(): string; // inherited property
  prop: number; // own property
}
const obj: MyInterface = { // 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.

15.8 Sources of this chapter

- TypeScript Handbook
- TypeScript Language Specification

Chapter 16

Class definitions in TypeScript

Contents

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.

16.1 Cheat sheet: classes in plain JavaScript

This section is a cheat sheet for class definitions in plain JavaScript.

16.1.1 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.

16.1.2 Modifier: `static`

```
class MyClass2 {

  static staticPublicField = 1;

  static staticPublicMethod() {
    return 2;
  }
}

assert.equal(MyClass2.staticPublicField, 1);
assert.equal(MyClass2.staticPublicMethod(), 2);
```

16.1.3 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);
    }
  }
  MyClass3.accessPrivateMembers();
```

Warning for JavaScript:

- Support for private methods is currently quite limited.
- Private fields have broader, but also limited, support.

TypeScript has been supporting private fields since version 3.8 but does not currently support private methods.

16.1.4 Modifiers for accessors: `get` (getter) and `set` (setter)

Roughly, accessors are methods that are invoked by accessing properties. There are two kinds of accessors: getters and setters.

```
class MyClass5 {
  #name = 'Rumpelstiltskin';

  /** Prototype getter */
  get name() {
    return this.#name;
  }

  /** Prototype setter */
  set name(value) {
    this.#name = value;
  }
}
const inst5 = new MyClass5();
assert.equal(inst5.name, 'Rumpelstiltskin'); // getter
inst5.name = 'Queen'; // setter
assert.equal(inst5.name, 'Queen'); // getter
```

16.1.5 Modifier for methods: `*` (generator)

```
class MyClass6 {
  * publicPrototypeGeneratorMethod() {
    yield 'hello';
    yield 'world';
  }
}

const inst6 = new MyClass6();
```

```
assert.deepEqual(
  [...inst6.publicPrototypeGeneratorMethod()],
  ['hello', 'world']);
```

16.1.6 Modifier for methods: async

```
class MyClass7 {
  async publicPrototypeAsyncMethod() {
    const result = await Promise.resolve('abc');
    return result + result;
  }
}

const inst7 = new MyClass7();
inst7.publicPrototypeAsyncMethod()
  .then(result => assert.equal(result, 'abcabc'));
```

16.1.7 Computed class member names

```
const publicInstanceFieldKey = Symbol('publicInstanceFieldKey');
const publicPrototypeMethodKey = Symbol('publicPrototypeMethodKey');

class MyClass8 {

  [publicInstanceFieldKey] = 1;

  [publicPrototypeMethodKey]() {
    return 2;
  }
}

const inst8 = new MyClass8();
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).

16.1.8 Combinations of modifiers

Fields (no level means that a construct exists at the instance level):

Level	Visibility
(instance)	
(instance)	#

Level	Visibility
`static`	
`static`	`#`

Methods (no level means that a construct exists at the prototype level):

Level	Accessor	Async	Generator	Visibility
(prototype)				
(prototype)	`get`			
(prototype)	`set`			
(prototype)		`async`		
(prototype)			`*`	
(prototype)		`async`	`*`	
(prototype-associated)				`#`
(prototype-associated)	`get`			`#`
(prototype-associated)	`set`			`#`
(prototype-associated)		`async`		`#`
(prototype-associated)			`*`	`#`
(prototype-associated)		`async`	`*`	`#`
`static`				
`static`	`get`			
`static`	`set`			
`static`		`async`		
`static`			`*`	
`static`		`async`	`*`	
`static`				`#`
`static`	`get`			`#`
`static`	`set`			`#`
`static`		`async`		`#`
`static`			`*`	`#`
`static`		`async`	`*`	`#`

Limitations of methods:

- Accessors can't be async or generators.

16.1.9 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);
```

Fig. 16.1 shows what the prototype chains look like that are created by `ClassA` and `ClassB`.

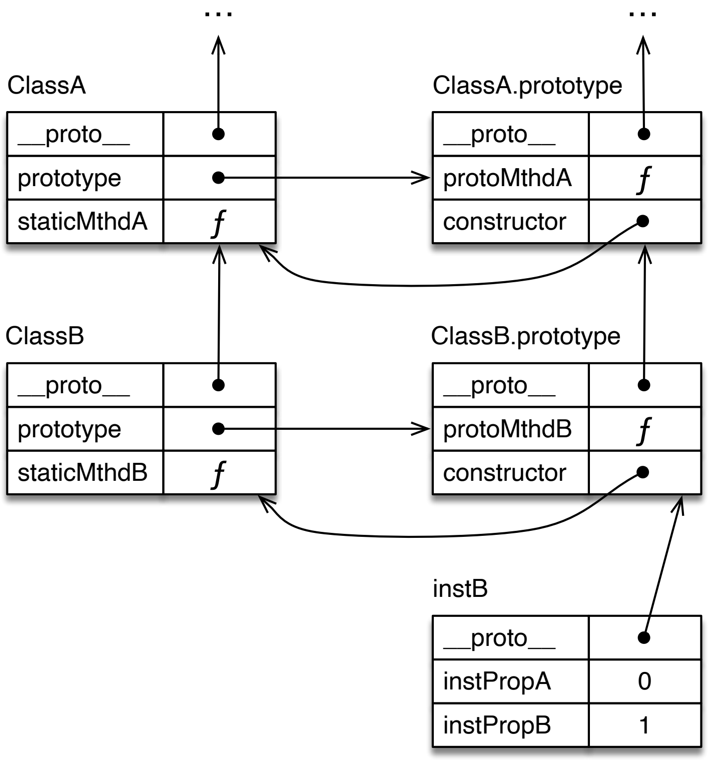


Figure 16.1: The classes `ClassA` and `ClassB` create two prototype chains: One for classes (left-hand side) and one for instances (right-hand side).

16.1.10 More information on class definitions in plain JavaScript

- Public fields, private fields, private methods/getters/setters (blog post)
- All remaining JavaScript class features (chapter in "JavaScript for impatient programming")

16.2 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.

16.2.1 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'. (2341)
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}!`;
  }
}
```

16.2.2 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),
  []);
```

The compilation result is much more complicated now (slightly simplified):

```
var __classPrivateFieldSet = function (receiver, privateMap, value) {
  if (!privateMap.has(receiver)) {
    throw new TypeError(
      'attempted to set private field on non-instance');
  }
  privateMap.set(receiver, value);
  return value;
};

// Omitted: __classPrivateFieldGet

var _name = new WeakMap();
class Person {
  constructor(name) {
    // Add an entry for this instance to _name
    _name.set(this, void 0);

    // Now we can use the helper function:
    __classPrivateFieldSet(this, _name, name);
  }
  // ···
}
```

This code uses a common technique for keeping instance data private:

- Each WeakMap implements one private field.
- It associates each instance with one piece of private data.

More information on this topic: see "JavaScript for impatient programmers".

16.2.3 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.
- Upside 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.

16.2.4 Protected properties

Private fields and private properties can't be accessed in subclasses (line A):

```
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;
  }
  sayHello() {
    // @ts-expect-error: Property 'name' is private and only
    // accessible within class 'PrivatePerson'. (2341)
    return `Hello ${this.name} from ${this.company}!`; // (A)
  }
}
```

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;
  }
  sayHello() {
    return `Hello ${this.name} from ${this.company}!`; // OK
  }
}
```

16.3 Private constructors

Constructors can be private, too. 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;
  }
}
DataContainer.create()
  .then(dc => assert.equal(
    dc.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`.

16.4 Initializing instance properties

16.4.1 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;
  }
}
```

16.4.1.1 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`.

16.4.2 Making constructor parameters `public`, `private`, or `protected`

If we use the keyword `public` for a constructor parameter, then TypeScript does two things for us:

- It declares a public instance property with the same name.
- It assigns the parameter to that instance property.

Therefore, the following two classes are equivalent:

```
class Point1 {
  constructor(public x: number, public y: number) {
  }
}

class Point2 {
  x: number;
  y: number;
  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
}
```

If we use `private` or `protected` instead of `public`, then the corresponding instance properties are private or protected (not public).

16.5 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.

Chapter 17

Class-related types

Contents

In this chapter about TypeScript, we examine types related to classes and their instances.

17.1 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);
```

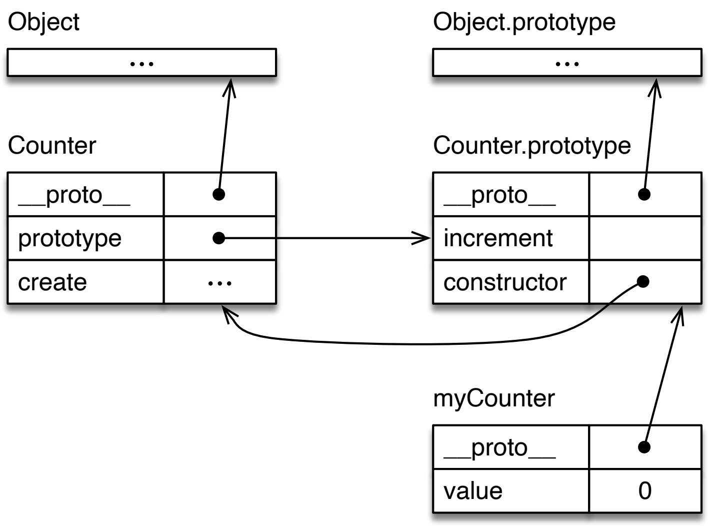


Figure 17.1: Objects created by class `Counter`. Left-hand side: the class and its superclass `Object`. Right-hand side: The instance `myCounter`, the prototype properties of `Counter`, and the prototype methods of the superclass `Object`..

The diagram in fig. 17.1 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.

17.2 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:

- 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.

17.3 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.

17.3.1 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: any): JsonInstance;
  }

  // Converting instances to JSON
  interface JsonInstance {
    toJson(): any;
  }
```

We use these interfaces in the following code:

```
  class Person implements JsonInstance {
    static fromJson(json: any): Person {
      if (typeof json !== 'string') {
        throw new TypeError(json);
      }
      return new Person(json);
    }
    name: string;
    constructor(name: string) {
      this.name = name;
    }
    toJson(): any {
      return this.name;
    }
  }
```

This is how we can check right away if class `Person` (as an object) implements the interface `JsonStatic`:

```
// Assign the class to a type-annotated variable
const personImplementsJsonStatic: JsonStatic = Person;
```

The following way of making this check may seem like a good idea:

```
const Person: JsonStatic = class implements JsonInstance {
  // ···
};
```

However, that doesn't really work:

- We can't `new`-call `Person` because `JsonStatic` does not have a construct signature.
- If `Person` has static properties beyond `.fromJson()`, TypeScript won't let us access them.

17.3.2 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 class `Object` itself:

```
/**
 * Provides functionality common to all JavaScript objects.
 */
declare var Object: ObjectConstructor;

interface ObjectConstructor {
  new(value?: any): Object;
  (): any;
  (value: any): any;

  /** A reference to the prototype for a class of objects. */
  readonly prototype: Object;
  /**
   * Returns the prototype of an object.
   * @param o The object that references the prototype.
   */
  getPrototypeOf(o: any): any;

}
```

On the other hand, interface `Object` is for instances of `Object`:

```
interface Object {
  /** The initial value of Object.prototype.constructor is the standard built-in Object
  constructor: Function;

  /** Returns a string representation of an object. */
  toString(): string;
}
```

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.

17.4 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];
}
```

17.4.1 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.

17.4.1.1 Switching off structural typing

We can make the two groups of objects incompatible by adding private properties:

```
class Color {
  name: string;
  private branded = true;
  constructor(name: string) {
    this.name = name;
  }
}
class Person {
  name: string;
  private branded = true;
  constructor(name: string) {
    this.name = name;
  }
}

const person: Person = new Person('Jane');

// @ts-expect-error: Type 'Person' is not assignable to type 'Color'.
//   Types have separate declarations of a private property
//   'branded'. (2322)
const color: Color = person;
```

The private properties switch off structural typing in this case.

17.5 Further reading

- Chapter "Prototype chains and classes" in "JavaScript for impatient programmers"

Chapter 18

Types for classes as values

Contents

In this chapter, we explore classes as values:

- What types should we use for such values?
- What are the use cases for these types?

18.1 Types for specific classes

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: ???, x: number, y: number) {
  return new PointClass(x, y);
}
```

What type should we use for the parameter `PointClass` if we want it to be `Point` or a subclass?

18.2 The type operator `typeof`

In §7.7 "The two language levels: dynamic vs. static", 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` (another bit of TypeScript syntax that also exists in JavaScript). `typeof v` stands for the type of the dynamic(!) value `v`.

```
function createPoint(PointClass: typeof Point, x: number, y: number) { // (A)
  return new PointClass(x, y);
}

// %inferred-type: Point
const point = createPoint(Point, 3, 6);
assert.ok(point instanceof Point);
```

18.2.1 Constructor type literals

A *constructor type literal* is a function type literal with a prefixed `new` (line A). The prefix indicates that `PointClass` is a function that must be invoked via `new`.

```
function createPoint(
  PointClass: new (x: number, y: number) => Point, // (A)
  x: number, y: number
) {
  return new PointClass(x, y);
}
```

18.2.2 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);
}
```

18.3 A generic type for classes: `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`.

18.3.1 Example: creating instances

`Class<T>` enables us to write a generic version of `createPoint()`:

```
function createInstance<T>(AnyClass: Class<T>, ...args: any[]): T {
  return new AnyClass(...args);
}
```

`createInstance()` is used as follows:

```
class Person {
  constructor(public name: string) {}
}

// %inferred-type: Person
const jane = createInstance(Person, 'Jane');
```

`createInstance()` is the `new` operator, implemented via a function.

18.3.2 Example: casting with runtime checks

We can use `Class<T>` to implement casting:

```
function cast<T>(AnyClass: Class<T>, obj: any): T {
  if (! (obj instanceof AnyClass)) {
    throw new Error(`Not an instance of ${AnyClass.name}: ${obj}`)
  }
  return obj;
}
```

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 {
  // %inferred-type: any
  const parsed = JSON.parse(jsonObjectStr);
  return cast(Object, parsed);
}
```

18.3.3 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<any, any>();
  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: any) {
    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/);

// %inferred-type: RegExp
const re = map.get(RegExp);

// Static and dynamic error!
assert.throws(
  // @ts-expect-error: Argument of type '"abc"' is not assignable
  // to parameter of type 'Date'.
  () => map.set(Date, 'abc'));
```

18.3.4 Pitfall: Class<T> does not match abstract classes

We cannot use abstract classes when `Class<T>` is expected:

```
abstract class Shape {
}
class Circle extends Shape {
    // ···
}

// @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. (2322)
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 (GitHub issue with more information).

This is a workaround:

```
type Class2<T> = Function & {prototype: T};

const shapeClasses2: Array<Class2<Shape>> = [Circle, Shape];
```

Downsides of this approach:

- Slightly confusing.
- Values that have this type can't be used for `instanceof` checks (as right-hand-side operands).

Chapter 19

Typing Arrays

Contents

In this chapter, we examine how Arrays can be typed in TypeScript.

19.1 Roles of Arrays

Arrays can play the following roles in JavaScript (either one or a mix of them):

- Lists: All elements have the same type. The length of the Array varies.
- Tuple: The length of the Array is fixed. The elements do not necessarily have the same type.

TypeScript accommodates these two roles by offering various ways of typing arrays. We will look at those next.

19.2 Ways of typing Arrays

19.2.1 Array role "list": Array type literals vs. interface type `Array`

An Array type literal consists of the element type followed by `[]`. In the following code, the Array type literal is `string[]`:

```
// Each Array element has the type `string`:
const myStringArray: string[] = ['fee', 'fi', 'fo', 'fum'];
```

An Array type literal is a shorthand for using the global generic interface type `Array`:

```
const myStringArray: Array<string> = ['fee', 'fi', 'fo', 'fum'];
```

If the element type is more complicated, we need parentheses for Array type literals:

```
(number|string)[]
(() => boolean)[]
```

The parameterized type `Array` works better in this case:

```
Array<number|string>
Array<() => boolean>
```

19.2.2 Array role "tuple": tuple type literals

If the Array has a fixed length and each element has a different, fixed type that depends on its position, then we can use tuple type literals such as `[string, string, boolean]`:

```
const yes: [string, string, boolean] = ['oui', 'sí', true];
```

19.2.3 Objects that are also Array-ish: interfaces with index signatures

If an interface has only an index signature, we can use it for Arrays:

```
interface StringArray {
  [index: number]: string;
}
const strArr: StringArray = ['Huey', 'Dewey', 'Louie'];
```

An interface that has both an index signature and property signatures, only works for objects (because indexed elements and properties need to be defined at the same time):

```
interface FirstNamesAndLastName {
  [index: number]: string;
  lastName: string;
}

const ducks: FirstNamesAndLastName = {
  0: 'Huey',
  1: 'Dewey',
  2: 'Louie',
  lastName: 'Duck',
};
```

19.3 Pitfall: type inference doesn't always get Array types right

19.3.1 Inferring types of Arrays is difficult

Due to the two roles of Arrays, it is impossible for TypeScript to always guess the right type. As an example, consider the following Array literal that is assigned to the variable `fields`:

```
const fields: Fields = [
  ['first', 'string', true],
  ['last', 'string', true],
  ['age', 'number', false],
];
```

What is the best type for `fields`? The following are all reasonable choices:

```
type Fields = Array<[string, string, boolean]>;
```

```
type Fields = Array<[string, ('string'|'number'), boolean]>;
```

```
type Fields = Array<Array<string|boolean>>;
```

```
type Fields = [
  [string, string, boolean],
  [string, string, boolean],
  [string, string, boolean],
];
```

```
type Fields = [
  [string, 'string', boolean],
  [string, 'string', boolean],
  [string, 'number', boolean],
];
```

```
type Fields = [
  Array<string|boolean>,
  Array<string|boolean>,
  Array<string|boolean>,
];
```

19.3.2 Type inference for non-empty Array literals

When we use non-empty Array literals, TypeScript's default is to infer list types (not tuple types):

```
// %inferred-type: (string | number)[]
const arr = [123, 'abc'];
```

Alas, that's not always what we want:

```
function func(p: [number, number]) {
  return p;
}
// %inferred-type: number[]
const pair1 = [1, 2];

// @ts-expect-error: Argument of type 'number[]' is not assignable to
// parameter of type '[number, number]'. [...]
func(pair1);
```

We can fix this by adding a type annotation to the `const` declaration, which avoids type inference:

```
const pair2: [number, number] = [1, 2];
func(pair2); // OK
```

19.3.3 Type inference for empty Array literals

If we initialize a variable with an empty Array literal, then TypeScript initially infers the type `any[]` and incrementally updates that type as we make changes:

```
// %inferred-type: any[]
const arr1 = [];

arr1.push(123);
// %inferred-type: number[]
arr1;

arr1.push('abc');
// %inferred-type: (string | number)[]
arr1;
```

Note that the initial inferred type isn't influenced by what happens later.

If we use assignment instead of `.push()`, things work the same:

```
// %inferred-type: any[]
const arr1 = [];

arr1[0] = 123;
// %inferred-type: number[]
arr1;

arr1[1] = 'abc';
// %inferred-type: (string | number)[]
arr1;
```

In contrast, if the Array literal has at least one element, then the element type is fixed and doesn't change later:

```
// %inferred-type: number[]
const arr = [123];

// @ts-expect-error: Argument of type '"abc"' is not assignable to
// parameter of type 'number'. (2345)
arr.push('abc');
```

19.3.4 const assertions for Arrays and type inference

We can suffix an Array literal with a `const` assertion:

```
// %inferred-type: readonly ["igneous", "metamorphic", "sedimentary"]
const rockCategories =
  ['igneous', 'metamorphic', 'sedimentary'] as const;
```

We are declaring that `rockCategories` won't change. That has the following effects:

- The Array becomes `readonly` – we can't use operations that change it:

  ```
  // @ts-expect-error: Property 'push' does not exist on type
  // 'readonly ["igneous", "metamorphic", "sedimentary"]'. (2339)
  rockCategories.push('sand');
  ```

- TypeScript infers a tuple. Compare:

  ```
  // %inferred-type: string[]
  const rockCategories2 = ['igneous', 'metamorphic', 'sedimentary'];
  ```

- TypeScript infers literal types (`"igneous"` etc.) instead of more general types. That is, the inferred tuple type is not `[string, string, string]`.

Here are more examples of Array literals with and without `const` assertions:

```
// %inferred-type: readonly [1, 2, 3, 4]
const numbers1 = [1, 2, 3, 4] as const;
// %inferred-type: number[]
const numbers2 = [1, 2, 3, 4];

// %inferred-type: readonly [true, "abc"]
const booleanAndString1 = [true, 'abc'] as const;
// %inferred-type: (string | boolean)[]
const booleanAndString2 = [true, 'abc'];
```

19.3.4.1 Potential pitfalls of const assertions

There are two potential pitfalls with `const` assertions.

First, the inferred type is as narrow as possible. That causes an issue for `let`-declared variables: We cannot assign any tuple other than the one that we used for intialization:

```
let arr = [1, 2] as const;

arr = [1, 2]; // OK

// @ts-expect-error: Type '3' is not assignable to type '2'. (2322)
arr = [1, 3];
```

Second, tuples declared via `as const` can't be mutated:

```
let arr = [1, 2] as const;

// @ts-expect-error: Cannot assign to '1' because it is a read-only
// property. (2540)
arr[1] = 3;
```

That is neither an upside nor a downside, but we need to be aware that it happens.

19.4 Pitfall: TypeScript assumes indices are never out of bounds

Whenever we access an Array element via an index, TypeScript always assumes that the index is within range (line A):

```
const messages: string[] = ['Hello'];

// %inferred-type: string
const message = messages[3]; // (A)
```

Due to this assumption, the type of `message` is `string`. And not `undefined` or `undefined|string`, as we may have expected.

We do get an error if we use a tuple type:

```
const messages: [string] = ['Hello'];

// @ts-expect-error: Tuple type '[string]' of length '1' has no element
// at index '1'. (2493)
const message = messages[1];
```

`as const` would have had the same effect because it leads to a tuple type being inferred.

Chapter 20

Typing functions

Contents

This chapter explores static typing for functions in TypeScript.

in this chapter, "function" means "function or method or constructor"

In this chapter, most things that are said about functions (especially w.r.t. parameter handling), also apply to methods and constructors.

20.1 Defining statically typed functions

20.1.1 Function declarations

This is an example of a function declaration in TypeScript:

```
function repeat1(str: string, times: number): string { // (A)
  return str.repeat(times);
}
assert.equal(
  repeat1('*', 5), '*****');
```

- Parameters: If the compiler option `--noImplicitAny` is on (which it is if `--strict` is on), then the type of each parameter must be either inferrable or explicitly specified. (We'll take a closer look at inference later.) In this case, no inference is possible, which is why `str` and `times` have type annotations.

- Return value: By default, the return type of functions is inferred. That is usually good enough. In this case, we opted to explicitly specify that `repeat1()` has the return type `string` (last type annotation in line A).

20.1.2 Arrow functions

The arrow function version of `repeat1()` looks as follows:

```
const repeat2 = (str: string, times: number): string => {
  return str.repeat(times);
};
```

In this case, we can also use an expression body:

```
const repeat3 = (str: string, times: number): string =>
  str.repeat(times);
```

20.2 Types for functions

20.2.1 Function type signatures

We can define types for functions via function type signatures:

```
type Repeat = (str: string, times: number) => string;
```

The name of this type of function is `Repeat`. Among others, it matches all functions with:

- Two parameters whose types are `string` and `number`. We need to name parameters in function type signatures, but the names are ignored when checking if two function types are compatible.
- The return type `string`. Note that this time, the type is separated by an arrow and can't be omitted.

This type matches more functions. We'll learn which ones, when we explore the rules for *assignability* later in this chapter.

20.2.2 Interfaces with call signatures

We can also use interfaces to define function types:

```
interface Repeat {
  (str: string, times: number): string; // (A)
}
```

Note:

- The interface member in line A is a *call signature*. It looks similar to a method signature, but doesn't have a name.
- The type of the result is separated by a colon (not an arrow) and can't be omitted.

On one hand, interfaces are more verbose. On the other hand, they let us specify properties of functions (which is rare, but does happen):

```
interface Incrementor1 {
  (x: number): number;
  increment: number;
}
```

We can also specify properties via an intersection type (&) of a function signature type and an object literal type:

```
type Incrementor2 =
  (x: number) => number
  & { increment: number }
;
```

20.2.3 Checking if a callable value matches a function type

As an example, consider this scenario: A library exports the following function type.

```
type StringPredicate = (str: string) => boolean;
```

We want to define a function whose type is compatible with `StringPredicate`. And we want to check immediately if that's indeed the case (vs. finding out later when we use it for the first time).

20.2.3.1 Checking arrow functions

If we declare a variable via `const`, we can perform the check via a type annotation:

```
const pred1: StringPredicate = (str) => str.length > 0;
```

Note that we don't need to specify the type of parameter `str` because TypeScript can use `StringPredicate` to infer it.

20.2.3.2 Checking function declarations (simple)

Checking function declarations is more complicated:

```
function pred2(str: string): boolean {
  return str.length > 0;
}

// Assign the function to a type-annotated variable
const pred2ImplementsStringPredicate: StringPredicate = pred2;
```

20.2.3.3 Checking function declarations (extravagant)

The following solution is slightly over the top (i.e., don't worry if you don't fully understand it), but it demonstrates several advanced features:

```
function pred3(...[str]: Parameters<StringPredicate>)
  : ReturnType<StringPredicate> {
    return str.length > 0;
  }
```

- Parameters: We use `Parameters<>` to extract a tuple with the parameter types. The three dots declare a rest parameter, which collects all parameters in a tuple/Array. `[str]` destructures that tuple. (More on rest parameters later in this chapter.)
- Return value: We use `ReturnType<>` to extract the return type.

20.3 Parameters

20.3.1 When do parameters have to be type-annotated?

Recap: If `--noImplicitAny` is switched on (`--strict` switches it on), the type of each parameter must either be inferrable or explicitly specified.

In the following example, TypeScript can't infer the type of `str` and we must specify it:

```
function twice(str: string) {
  return str + str;
}
```

In line A, TypeScript can use the type `StringMapFunction` to infer the type of `str` and we don't need to add a type annotation:

```
type StringMapFunction = (str: string) => string;
const twice: StringMapFunction = (str) => str + str; // (A)
```

Here, TypeScript can use the type of `.map()` to infer the type of `str`:

```
assert.deepEqual(
  ['a', 'b', 'c'].map((str) => str + str),
  ['aa', 'bb', 'cc']);
```

This is the type of `.map()`:

```
interface Array<T> {
  map<U>(
    callbackfn: (value: T, index: number, array: T[]) => U,
    thisArg?: any
  ): U[];
  // ···
}
```

20.3.2 Optional parameters

In this section, we look at several ways in which we can allow parameters to be omitted.

20.3.2.1 Optional parameter: `str?: string`

If we put a question mark after the name of a parameter, that parameter becomes optional and can be omitted when calling the function:

```
function trim1(str?: string): string {
  // Internal type of str:
  // %inferred-type: string | undefined
  str;

  if (str === undefined) {
    return '';
  }
  return str.trim();
}

// External type of trim1:
// %inferred-type: (str?: string | undefined) => string
trim1;
```

This is how `trim1()` can be invoked:

```
assert.equal(
  trim1('\n  abc \t'), 'abc');

assert.equal(
  trim1(), '');

// `undefined` is equivalent to omitting the parameter
assert.equal(
  trim1(undefined), '');
```

20.3.2.2 Union type: `str: undefined|string`

Externally, parameter `str` of `trim1()` has the type `string|undefined`. Therefore, `trim1()` is mostly equivalent to the following function.

```
function trim2(str: undefined|string): string {
  // Internal type of str:
  // %inferred-type: string | undefined
  str;

  if (str === undefined) {
    return '';
  }
  return str.trim();
}

// External type of trim2:
// %inferred-type: (str: string | undefined) => string
trim2;
```

The only way in which `trim2()` is different from `trim1()` is that the parameter can't be omitted in function calls (line A). In other words: We must be explicit when omitting a parameter whose type is `undefined|T`.

```
  assert.equal(
    trim2('\n  abc \t'), 'abc');

  // @ts-expect-error: Expected 1 arguments, but got 0. (2554)
  trim2(); // (A)

  assert.equal(
    trim2(undefined), ''); // OK!
```

20.3.2.3 Parameter default value: `str = ''`

If we specify a parameter default value for `str`, we don't need to provide a type annotation because TypeScript can infer the type:

```
  function trim3(str = ''): string {
    // Internal type of str:
    // %inferred-type: string
    str;

    return str.trim();
  }

  // External type of trim2:
  // %inferred-type: (str?: string) => string
  trim3;
```

Note that the internal type of `str` is `string` because the default value ensures that it is never `undefined`.

Let's invoke `trim3()`:

```
  assert.equal(
    trim3('\n  abc \t'), 'abc');
```

```
// Omitting the parameter triggers the parameter default value:
assert.equal(
  trim3(), '');

// `undefined` is allowed and triggers the parameter default value:
assert.equal(
  trim3(undefined), '');
```

20.3.2.4 Parameter default value plus type annotation

We can also specify both a type and a default value:

```
function trim4(str: string = ''): string {
  return str.trim();
}
```

20.3.3 Rest parameters

20.3.3.1 Rest parameters with Array types

A rest parameter collects all remaining parameters in an Array. Therefore, its static type is usually an Array. In the following example, `parts` is a rest parameter:

```
function join(separator: string, ...parts: string[]) {
  return parts.join(separator);
}
assert.equal(
  join('-', 'state', 'of', 'the', 'art'),
  'state-of-the-art');
```

20.3.3.2 Rest parameters with tuple types

The next example demonstrates two features:

- We can use tuple types such as `[string, number]` for rest parameters.
- We can destructure rest parameters (not just normal parameters).

```
function repeat1(...[str, times]: [string, number]): string {
  return str.repeat(times);
}
```

`repeat1()` is equivalent to the following function:

```
function repeat2(str: string, times: number): string {
  return str.repeat(times);
}
```

20.3.4 Named parameters

Named parameters are a popular pattern in JavaScript where an object literal is used to give each parameter a name. That looks as follows:

```
assert.equal(
  padStart({str: '7', len: 3, fillStr: '0'}),
  '007');
```

In plain JavaScript, functions can use destructuring to access named parameter values. Alas, in TypeScript, we additionally have to specify a type for the object literal and that leads to redundancies:

```
function padStart({ str, len, fillStr = ' ' } // (A)
  : { str: string, len: number, fillStr: string }) { // (B)
  return str.padStart(len, fillStr);
}
```

Note that the destructuring (incl. the default value for `fillStr`) all happens in line A, while line B is exclusively about TypeScript.

It is possible to define a separate type instead of the inlined object literal type that we have used in line B. However, in most cases, I prefer not to do that because it slightly goes against the nature of parameters which are local and unique per function. If you prefer having less stuff in function heads, then that's OK, too.

20.3.5 `this` as a parameter (advanced)

Each ordinary function always has the implicit parameter `this` – which enables it to be used as a method in objects. Sometimes we need to specify a type for `this`. There is TypeScript-only syntax for this use case: One of the parameters of an ordinary function can have the name `this`. Such a parameter only exists at compile time and disappears at runtime.

As an example, consider the following interface for DOM event sources (in a slightly simplified version):

```
interface EventSource {
  addEventListener(
    type: string,
    listener: (this: EventSource, ev: Event) => any,
    options?: boolean | AddEventListenerOptions
  ): void;
  // ···
}
```

The `this` of the callback `listener` is always an instance of `EventSource`.

The next example demonstrates that TypeScript uses the type information provided by the `this` parameter to check the first argument of `.call()` (line A and line B):

```
function toIsoString(this: Date): string {
    return this.toISOString();
}

// @ts-expect-error: Argument of type '"abc"' is not assignable to
// parameter of type 'Date'. (2345)
assert.throws(() => toIsoString.call('abc')); // (A) error
```

```
toIsoString.call(new Date()); // (B) OK
```

Additionally, we can't invoke `toIsoString()` as a method of an object `obj` because then its receiver isn't an instance of `Date`:

```
const obj = { toIsoString };
// @ts-expect-error: The 'this' context of type
// '{ toIsoString: (this: Date) => string; }' is not assignable to
// method's 'this' of type 'Date'. [...]
assert.throws(() => obj.toIsoString()); // error
obj.toIsoString.call(new Date()); // OK
```

20.4 Overloading (advanced)

Sometimes a single type signature does not adequately describe how a function works.

20.4.1 Overloading function declarations

Consider function `getFullName()` which we are calling in the following example (line A and line B):

```
interface Customer {
  id: string;
  fullName: string;
}
const jane = {id: '1234', fullName: 'Jane Bond'};
const lars = {id: '5678', fullName: 'Lars Croft'};
const idToCustomer = new Map<string, Customer>([
  ['1234', jane],
  ['5678', lars],
]);

assert.equal(
  getFullName(idToCustomer, '1234'), 'Jane Bond'); // (A)

assert.equal(
  getFullName(lars), 'Lars Croft'); // (B)
```

How would we implement `getFullName()`? The following implementation works for the two function calls in the previous example:

```
function getFullName(
  customerOrMap: Customer | Map<string, Customer>,
  id?: string
): string {
  if (customerOrMap instanceof Map) {
    if (id === undefined) throw new Error();
    const customer = customerOrMap.get(id);
    if (customer === undefined) {
      throw new Error('Unknown ID: ' + id);
    }
    customerOrMap = customer;
  } else {
    if (id !== undefined) throw new Error();
  }
  return customerOrMap.fullName;
}
```

However, with this type signature, function calls are legal at compile time that produce runtime errors:

```
assert.throws(() => getFullName(idToCustomer)); // missing ID
assert.throws(() => getFullName(lars, '5678')); // ID not allowed
```

The following code fixes these issues:

```
function getFullName(customerOrMap: Customer): string; // (A)
function getFullName( // (B)
  customerOrMap: Map<string, Customer>, id: string): string;
function getFullName( // (C)
  customerOrMap: Customer | Map<string, Customer>,
  id?: string
): string {
  // ···
}

// @ts-expect-error: Argument of type 'Map<string, Customer>' is not
// assignable to parameter of type 'Customer'. [...]
getFullName(idToCustomer); // missing ID

// @ts-expect-error: Argument of type '{ id: string; fullName: string; }'
// is not assignable to parameter of type 'Map<string, Customer>'.
// [...]
getFullName(lars, '5678'); // ID not allowed
```

What is going on here? The type signature of `getFullName()` is overloaded:

- The actual implementation starts in line C. It is the same as in the previous example.
- In line A and line B there are the two type signatures (function heads without bodies) that can be used for `getFullName()`. The type signature of the actual implementation cannot be used!

My advice is to only use overloading when it can't be avoided. One alternative is to split an overloaded function into multiple functions with different names – for example:

- `getFullName()`
- `getFullNameViaMap()`

20.4.2 Overloading via interfaces

In interfaces, we can have multiple, different call signatures. That enables us to use the interface `GetFullName` for overloading in the following example:

```
interface GetFullName {
  (customerOrMap: Customer): string;
  (customerOrMap: Map<string, Customer>, id: string): string;
}

const getFullName: GetFullName = (
  customerOrMap: Customer | Map<string, Customer>,
  id?: string
): string => {
  if (customerOrMap instanceof Map) {
    if (id === undefined) throw new Error();
    const customer = customerOrMap.get(id);
    if (customer === undefined) {
      throw new Error('Unknown ID: ' + id);
    }
    customerOrMap = customer;
  } else {
    if (id !== undefined) throw new Error();
  }
  return customerOrMap.fullName;
}
```

20.4.3 Overloading on string parameters (event handling etc.)

In the next example, we overload and use string literal types (such as `'click'`). That allows us to change the type of parameter `listener` depending on the value of parameter `type`:

```
function addEventListener(elem: HTMLElement, type: 'click',
  listener: (event: MouseEvent) => void): void;
function addEventListener(elem: HTMLElement, type: 'keypress',
  listener: (event: KeyboardEvent) => void): void;
function addEventListener(elem: HTMLElement, type: string,  // (A)
  listener: (event: any) => void): void {
    elem.addEventListener(type, listener); // (B)
  }
```

In this case, it is relatively difficult to get the types of the implementation (starting in line A) right, so that the statement in the body (line B) works. As a last resort, we can always use the type `any`.

20.4.4 Overloading methods

20.4.4.1 Overloading concrete methods

The next example demonstrates overloading of methods: Method `.add()` is overloaded.

```
class StringBuilder {
  #data = '';

  add(num: number): this;
  add(bool: boolean): this;
  add(str: string): this;
  add(value: any): this {
    this.#data += String(value);
    return this;
  }

  toString() {
    return this.#data;
  }
}

const sb = new StringBuilder();
sb
  .add('I can see ')
  .add(3)
  .add(' monkeys!')
;
assert.equal(
  sb.toString(), 'I can see 3 monkeys!')
```

20.4.4.2 Overloading interface methods

The type definition for `Array.from()` is an example of an overloaded interface method:

```
interface ArrayConstructor {
  from<T>(arrayLike: ArrayLike<T>): T[];
  from<T, U>(
    arrayLike: ArrayLike<T>,
    mapfn: (v: T, k: number) => U,
    thisArg?: any
  ): U[];
}
```

- In the first signature, the returned Array has the same element type as the parameter.
- In the second signature, the elements of the returned Array have the same type as the result of `mapfn`. This version of `Array.from()` is similar to `Array.prototype.map()`.

20.5 Assignability (advanced)

In this section we look at the type compatibility rules for *assignability*: Can functions of type `Src` be transferred to storage locations (variables, object properties, parameters, etc.)

of type `Trg`?

Understanding assignability helps us answer questions such as:

- Given the function type signature of a formal parameter, which functions can be passed as actual parameters in function calls?
- Given the function type signature of a property, which functions can be assigned to it?

20.5.1 The rules for assignability

In this subsection, we examine general rules for assignability (including the rules for functions). In the next subsection, we explore what those rules mean for functions.

A type `Src` is assignable to a type `Trg` if one of the following conditions is true:

- `Src` and `Trg` are identical types.
- `Src` or `Trg` is the `any` type.
- `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` and `Trg` are function types and:
  - `Trg` has a rest parameter or the number of required parameters of `Src` is less than or equal to the total number of parameters of `Trg`.
  - For parameters that are present in both signatures, each parameter type in `Trg` is assignable to the corresponding parameter type in `Src`.
  - The return type of `Trg` is `void` or the return type of `Src` is assignable to the return type of `Trg`.
- (Remaining conditions omitted.)

20.5.2 Consequences of the assignment rules for functions

In this subsection, we look at what the assignment rules mean for the following two functions `targetFunc` and `sourceFunc`:

```
const targetFunc: Trg = sourceFunc;
```

20.5.2.1 Types of parameters and results

- Target parameter types must be assignable to corresponding source parameter types.
  - Why? Anything that the target accepts must also be accepted by the source.
- The source return type must be assignable to target return type.
  - Why? Anything that the source returns must be compatible with the expectations set by the target.

Example:

```
const trg1: (x: RegExp) => Object = (x: Object) => /abc/;
```

The following example demonstrates that if the target return type is `void`, then the source return type doesn't matter. Why is that? `void` results are always ignored in TypeScript.

```
const trg2: () => void = () => new Date();
```

20.5.2.2 Numbers of parameters

The source must not have more parameters than the target:

```
// @ts-expect-error: Type '(x: string) => string' is not assignable to
// type '() => string'. (2322)
const trg3: () => string = (x: string) => 'abc';
```

The source can have fewer parameters than the target:

```
const trg4: (x: string) => string = () => 'abc';
```

Why is that? The target specifies the expectations for the source: It must accept the parameter `x`. Which it does (but it ignores it). This permissiveness enables:

```
['a', 'b'].map(x => x + x)
```

The callback for `.map()` only has one of the three parameters that are mentioned in the type signature of `.map()`:

```
map<U>(
  callback: (value: T, index: number, array: T[]) => U,
  thisArg?: any
): U[];
```

20.6 Further reading and sources of this chapter

- TypeScript Handbook
- TypeScript Language Specification
- Chapter "Callable values" in "JavaScript for impatient programmers"

Part IV

Dealing with ambiguous types

Chapter 21

The top types any and unknown

Contents

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.

21.1 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 §11 "What is a type in TypeScript? Two perspectives"), `any` and `unknown` are sets that contain all values. As an aside, TypeScript also has the *bottom type* `never`, which is the empty set.

21.2 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.

21.2.1 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.

21.2.2 Example: `String()`

The function `String()`, which converts arbitrary values to strings, has the following type signature:

```
interface StringConstructor {
  (value?: any): string; // call signature
  // ···
}
```

21.3 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.

Where `any` allows us to do anything, `unknown` is much more restrictive.

Before we can perform any operation on values of type `unknown`, we must first narrow their types via:

- Type assertions:

  ```
  function func(value: unknown) {
    // @ts-expect-error: Object is of type 'unknown'.
    value.toFixed(2);

    // Type assertion:
    (value as number).toFixed(2); // OK
  }
  ```

- Equality:

  ```
  function func(value: unknown) {
    // @ts-expect-error: Object is of type 'unknown'.
    value * 5;

    if (value === 123) { // equality
      // %inferred-type: 123
      value;

      value * 5; // OK
    }
  }
  ```

- Type guards:

  ```
  function func(value: unknown) {
    // @ts-expect-error: Object is of type 'unknown'.
    value.length;

    if (typeof value === 'string') { // type guard
      // %inferred-type: string
      value;

      value.length; // OK
    }
  }
  ```

- Assertion functions:

  ```
  function func(value: unknown) {
    // @ts-expect-error: Object is of type 'unknown'.
    value.test('abc');

    assertIsRegExp(value);

    // %inferred-type: 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 22

Type assertions (related to casting)

Contents

This chapter is about *type assertions* in TypeScript, which are related to type casts in other languages and performed via the `as` operator.

22.1 Type assertions

A type assertion lets us override a static type that TypeScript has computed for a value. That is useful for working around limitations of the type system.

Type assertions are related to type casts in other languages, but they don't throw exceptions and don't do anything at runtime (they do perform a few minimal checks statically).

```
const data: object = ['a', 'b', 'c']; // (A)

// @ts-expect-error: Property 'length' does not exist on type 'object'.
data.length; // (B)

assert.equal(
  (data as Array<string>).length, 3); // (C)
```

Comments:

- In line A, we widen the type of the Array to `object`.
- In line B, we see that this type doesn't let us access any properties (details).
- In line C, we use a type assertion (the operator `as`) to tell TypeScript that `data` is an Array. Now we can access property `.length`.

Type assertions are a last resort and should be avoided as much as possible. They (temporarily) remove the safety net that the static type system normally gives us.

Note that, in line A, we also overrode TypeScript's static type. But we did it via a type annotation. This way of overriding is much safer than type assertions because we are much more constrained: TypeScript's type must be assignable to the type of the annotation.

22.1.1 Alternative syntax for type assertions

TypeScript has an alternative "angle-bracket" syntax for type assertions:

```
<Array<string>>data
```

I recommend avoiding this syntax. It has grown out of style and is not compatible with React JSX code (in `.tsx` files).

22.1.2 Example: asserting an interface

In order to access property `.name` of an arbitrary object `obj`, we temporarily change the static type of `obj` to `Named` (line A and line B).

```
interface Named {
  name: string;
}
function getName(obj: object): string {
  if (typeof (obj as Named).name === 'string') { // (A)
    return (obj as Named).name; // (B)
  }
  return '(Unnamed)';
}
```

22.1.3 Example: asserting an index signature

In the following code (line A), we use the type assertion `as Dict`, so that we can access the properties of a value whose inferred type is `object`. That is, we are overriding the static type `object` with the static type `Dict`.

```
type Dict = {[k:string]: any};

function getPropertyValue(dict: unknown, key: string): any {
  if (typeof dict === 'object' && dict !== null && key in dict) {
    // %inferred-type: object
    dict;

    // @ts-expect-error: Element implicitly has an 'any' type because
    // expression of type 'string' can't be used to index type '{}'.
    // [...]
    dict[key];

    return (dict as Dict)[key]; // (A)
  } else {
    throw new Error();
  }
}
```

22.2 Constructs related to type assertions

22.2.1 Non-nullish assertion operator (postfix !)

If a value's type is a union that includes the types `undefined` or `null`, the *non-nullish assertion operator* (or *non-null assertion operator*) removes these types from the union. We are telling TypeScript: "This value can't be `undefined` or `null`." As a consequence, we can perform operations that are prevented by the types of these two values – for example:

```
const theName = 'Jane' as (null | string);

// @ts-expect-error: Object is possibly 'null'.
theName.length;

assert.equal(
  theName!.length, 4); // OK
```

22.2.1.1 Example – Maps: .get() after .has()

After we use the Map method `.has()`, we know that a Map has a given key. Alas, the result of `.get()` does not reflect that knowledge, which is why we have to use the nullish assertion operator:

```
function getLength(strMap: Map<string, string>, key: string): number {
  if (strMap.has(key)) {
    // We are sure x is not undefined:
    const value = strMap.get(key)!; // (A)
    return value.length;
  }
  return -1;
}
```

We can avoid the nullish assertion operator whenever the values of a Map can't be `undefined`. Then missing entries can be detected by checking if the result of `.get()` is `undefined`:

```
function getLength(strMap: Map<string, string>, key: string): number {
  // %inferred-type: string | undefined
  const value = strMap.get(key);
  if (value === undefined) { // (A)
```

```
      return -1;
    }

    // %inferred-type: string
    value;

    return value.length;
  }
```

22.2.2 Definite assignment assertions

If *strict property initialization* is switched on, we occasionally need to tell TypeScript that we do initialize certain properties – even though it thinks we don't.

This is an example where TypeScript complains even though it shouldn't:

```
class Point1 {
  // @ts-expect-error: Property 'x' has no initializer and is not definitely
  // assigned in the constructor.
  x: number;

  // @ts-expect-error: Property 'y' has no initializer and is not definitely
  // assigned in the constructor.
  y: number;

  constructor() {
    this.initProperties();
  }
  initProperties() {
    this.x = 0;
    this.y = 0;
  }
}
```

The errors go away if we use *definite assignment assertions* (exclamation marks) in line A and line B:

```
class Point2 {
  x!: number; // (A)
  y!: number; // (B)
  constructor() {
    this.initProperties();
  }
  initProperties() {
    this.x = 0;
    this.y = 0;
  }
}
```

Chapter 23

Type guards and assertion functions

Contents

In TypeScript, a value can have a type that is too general for some operations – for example, a union type. This chapter answers the following questions:

- What is *narrowing* of types?
  - Spoiler: *Narrowing* means changing the static type `T` of a storage location (such as a variable or a property) to a subset of `T`. For example, it is often useful to narrow the type `null|string` to the type `string`.
- What are *type guards* and *assertion functions* and how can we use them to narrow types?
  - Spoiler: `typeof` and `instanceof` are type guards.

23.1 When are static types too general?

To see how a static type can be too general, consider the following function `getScore()`:

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

The skeleton of `getScore()` looks as follows:

```
function getScore(value: number|string): number {
  // ···
}
```

Inside the body of `getScore()`, we don't know if the type of `value` `number` or `string`. Before we do, we can't really work with `value`.

23.1.1 Narrowing via `if` and type guards

The solution is to check the type of `value` at runtime, via `typeof` (line A and line B):

```
function getScore(value: number|string): number {
  if (typeof value === 'number') { // (A)
    // %inferred-type: number
    value;
    return value;
  }
  if (typeof value === 'string') { // (B)
    // %inferred-type: string
    value;
    return value.length;
  }
  throw new Error('Unsupported value: ' + value);
}
```

In this chapter, we interpret types as sets of values. (For more information on this interpretation and another one, see §11 "What is a type in TypeScript? Two perspectives".)

Inside the then-blocks starting in line A and line B, the static type of `value` changes, due to the checks we performed. We are now working with subsets of the original type `number|string`. This way of reducing the size of a type is called *narrowing*. Checking the result of `typeof` and similar runtime operations are called *type guards*.

Note that narrowing does not change the original type of `value`, it only makes it more specific as we pass more checks.

23.1.2 Narrowing via `switch` and a type guard

Narrowing also works if we use `switch` instead of `if`:

```
function getScore(value: number|string): number {
  switch (typeof value) {
    case 'number':
      // %inferred-type: number
      value;
      return value;
    case 'string':
      // %inferred-type: string
      value;
      return value.length;
    default:
      throw new Error('Unsupported value: ' + value);
  }
}
```

23.1.3 More cases of types being too general

These are more examples of types being too general:

- Nullable types:

  ```
  function func1(arg: null|string) {}
  function func2(arg: undefined|string) {}
  ```

- Discriminated unions:

  ```
  type Teacher = { kind: 'Teacher', teacherId: string };
  type Student = { kind: 'Student', studentId: string };
  type Attendee = Teacher | Student;

  function func3(attendee: Attendee) {}
  ```

- Types of optional parameters:

  ```
  function func4(arg?: string) {
    // %inferred-type: string | undefined
    arg;
  }
  ```

Note that these types are all union types!

23.1.4 The type `unknown`

If a value has the type `unknown`, we can do almost nothing with it and have to narrow its type first (line A):

```
function parseStringLiteral(stringLiteral: string): string {
  const result: unknown = JSON.parse(stringLiteral);
  if (typeof result === 'string') { // (A)
    return result;
  }
  throw new Error('Not a string literal: ' + stringLiteral);
}
```

In other words: The type `unknown` is too general and we must narrow it. In a way, `unknown` is also a union type (the union of all types).

23.2 Narrowing via built-in type guards

As we have seen, a *type guard* is an operation that returns either `true` or `false` – depending on whether its operand meets certain criteria at runtime. TypeScript's type inference supports type guards by narrowing the static type of an operand when the result is `true`.

23.2.1 Strict equality (===)

Strict equality works as a type guard:

```
function func(value: unknown) {
  if (value === 'abc') {
    // %inferred-type: "abc"
    value;
  }
}
```

For some union types, we can use === to differentiate between their components:

```
interface Book {
  title: null | string;
  isbn: string;
}

function getTitle(book: Book) {
  if (book.title === null) {
    // %inferred-type: null
    book.title;
    return '(Untitled)';
  } else {
    // %inferred-type: string
    book.title;
    return book.title;
    }
  }
```

Using `===` for including and `!===` for excluding a union type component only works if that component is a *singleton type* (a set with one member). The type `null` is a singleton type. Its only member is the value `null`.

23.2.2 `typeof`, `instanceof`, `Array.isArray`

These are three common built-in type guards:

```
function func(value: Function|Date|number[]) {
  if (typeof value === 'function') {
    // %inferred-type: Function
    value;
  }

  if (value instanceof Date) {
    // %inferred-type: Date
    value;
  }

  if (Array.isArray(value)) {
    // %inferred-type: number[]
    value;
  }
}
```

Note how the static type of `value` is narrowed inside the then-blocks.

23.2.3 Checking for distinct properties via the operator `in`

If used to check for distinct properties, the operator `in` is a type guard:

```
type FirstOrSecond =
  | {first: string}
  | {second: string};

function func(firstOrSecond: FirstOrSecond) {
  if ('second' in firstOrSecond) {
    // %inferred-type: { second: string; }
    firstOrSecond;
  }
}
```

Note that the following check would not have worked:

```
function func(firstOrSecond: FirstOrSecond) {
  // @ts-expect-error: Property 'second' does not exist on
  // type 'FirstOrSecond'. [...]
  if (firstOrSecond.second !== undefined) {
      // ···
    }
  }
```

The problem in this case is that, without narrowing, we can't access property `.second` of a value whose type is `FirstOrSecond`.

23.2.3.1 The operator `in` doesn't narrow non-union types

Alas, `in` only helps us with union types:

```
function func(obj: object) {
  if ('name' in obj) {
    // %inferred-type: object
    obj;

    // @ts-expect-error: Property 'name' does not exist on type 'object'.
    obj.name;
  }
}
```

23.2.4 Checking the value of a shared property (discriminated unions)

In a discriminated union, the components of a union type have one or more properties in common whose values are different for each component. Such properties are called *discriminants*.

Checking the value of a discriminant is a type guard:

```
type Teacher = { kind: 'Teacher', teacherId: string };
type Student = { kind: 'Student', studentId: string };
type Attendee = Teacher | Student;

function getId(attendee: Attendee) {
  switch (attendee.kind) {
    case 'Teacher':
      // %inferred-type: { kind: "Teacher"; teacherId: string; }
      attendee;
      return attendee.teacherId;
    case 'Student':
      // %inferred-type: { kind: "Student"; studentId: string; }
      attendee;
      return attendee.studentId;
    default:
      throw new Error();
  }
}
```

In the previous example, `.kind` is a discriminant: Each components of the union type `Attendee` has this property, with a unique value.

An `if` statement and equality checks work similarly to a `switch` statement:

```
function getId(attendee: Attendee) {
  if (attendee.kind === 'Teacher') {
    // %inferred-type: { kind: "Teacher"; teacherId: string; }
    attendee;
    return attendee.teacherId;
  } else if (attendee.kind === 'Student') {
    // %inferred-type: { kind: "Student"; studentId: string; }
    attendee;
    return attendee.studentId;
  } else {
    throw new Error();
  }
}
```

23.2.5 Narrowing dotted names

We can also narrow the types of properties (even of nested ones that we access via chains of property names):

```
type MyType = {
  prop?: number | string,
};
function func(arg: MyType) {
  if (typeof arg.prop === 'string') {
    // %inferred-type: string
    arg.prop; // (A)

    [].forEach((x) => {
      // %inferred-type: string | number | undefined
      arg.prop; // (B)
    });

    // %inferred-type: string
    arg.prop;

    arg = {};

    // %inferred-type: string | number | undefined
    arg.prop; // (C)
  }
}
```

Let's take a look at several locations in the previous code:

- Line A: We narrowed the type of `arg.prop` via a type guard.
- Line B: Callbacks may be executed much later (think of asynchronous code), which is why TypeScript undoes narrowing inside callbacks.
- Line C: The preceding assignment also undid narrowing.

23.2.6 Narrowing Array element types

23.2.6.1 The Array method `.every()` does not narrow

If we use `.every()` to check that all Array elements are non-nullish, TypeScript does not narrow the type of `mixedValues` (line A):

```
const mixedValues: ReadonlyArray<undefined|null|number> =
  [1, undefined, 2, null];

if (mixedValues.every(isNotNullish)) {
  // %inferred-type: readonly (number | null | undefined)[]
  mixedValues; // (A)
}
```

Note that `mixedValues` has to be read-only. If it weren't, another reference to it would statically allow us to push `null` into `mixedValues` inside the `if` statement. But that renders the narrowed type of `mixedValues` incorrect.

The previous code uses the following *user-defined type guard* (more on what that is soon):

```
function isNotNullish<T>(value: T): value is NonNullable<T> { // (A)
  return value !== undefined && value !== null;
}
```

`NonNullable<Union>` (line A) is a utility type that removes the types `undefined` and `null` from union type `Union`.

23.2.6.2 The Array method `.filter()` produces Arrays with narrower types

`.filter()` produces Arrays that have narrower types (i.e., it doesn't really narrow existing types):

```
// %inferred-type: (number | null | undefined)[]
const mixedValues = [1, undefined, 2, null];

// %inferred-type: number[]
const numbers = mixedValues.filter(isNotNullish);

function isNotNullish<T>(value: T): value is NonNullable<T> { // (A)
  return value !== undefined && value !== null;
}
```

Alas, we must use a type guard function directly – an arrow function with a type guard is not enough:

```
// %inferred-type: (number | null | undefined)[]
const stillMixed1 = mixedValues.filter(
  x => x !== undefined && x !== null);

// %inferred-type: (number | null | undefined)[]
const stillMixed2 = mixedValues.filter(
  x => typeof x === 'number');
```

23.3 User-defined type guards

TypeScript lets us define our own type guards – for example:

```
function isFunction(value: unknown): value is Function {
  return typeof value === 'function';
}
```

The return type `value is Function` is a *type predicate*. It is part of the type signature of `isFunction()`:

```
// %inferred-type: (value: unknown) => value is Function
isFunction;
```

A user-defined type guard must always return booleans. If `isFunction(x)` returns `true`, TypeScript narrows the type of the actual argument `x` to `Function`:

```
function func(arg: unknown) {
  if (isFunction(arg)) {
    // %inferred-type: Function
    arg; // type is narrowed
  }
}
```

Note that TypeScript doesn't care how we compute the result of a user-defined type guard. That gives us a lot of freedom w.r.t. the checks we use. For example, we could have implemented `isFunction()` as follows:

```
function isFunction(value: any): value is Function {
  try {
    value(); // (A)
    return true;
  } catch {
    return false;
  }
}
```

Alas, we have to use the type `any` for the parameter `value` because the type `unknown` does not let us make the function call in line A.

23.3.1 Example of a user-defined type guard: `isArrayWithInstancesOf()`

```
/**
 * This type guard for Arrays works similarly to `Array.isArray()`,
 * but also checks if all Array elements are instances of `T`.
 * As a consequence, the type of `arr` is narrowed to `Array<T>`
 * if this function returns `true`.
 *
 * Warning: This type guard can make code unsafe – for example:
 * We could use another reference to `arr` to add an element whose
 * type is not `T`. Then `arr` doesn't have the type `Array<T>`
 * anymore.
 */
function isArrayWithInstancesOf<T>(
  arr: any, Class: new (...args: any[])=>T)
  : arr is Array<T>
{
  if (!Array.isArray(arr)) {
    return false;
  }
  if (!arr.every(elem => elem instanceof Class)) {
    return false;
  }

  // %inferred-type: any[]
  arr; // (A)

  return true;
}
```

In line A, we can see that the inferred type of `arr` is *not* `Array<T>`, but our checks have ensured that it currently is. That's why we can return `true`. TypeScript trusts us and narrows to `Array<T>` when we use `isArrayWithInstancesOf()`:

```
const value: unknown = {};
if (isArrayWithInstancesOf(value, RegExp)) {
  // %inferred-type: RegExp[]
  value;
}
```

23.3.2 Example of a user-defined type guard: `isTypeof()`

23.3.2.1 A first attempt

This is a first attempt to implement `typeof` in TypeScript:

```
/**
 * An implementation of the `typeof` operator.
 */
function isTypeof<T>(value: unknown, prim: T): value is T {
  if (prim === null) {
    return value === null;
  }
  return value !== null && (typeof prim) === (typeof value);
}
```

Ideally, we'd be able to specify the expected type of `value` via a string (i.e., one of the results of `typeof`). But then we would have to derive the type `T` from that string and it's not immediately obvious how to do that (there is a way, as we'll see soon). As a workaround, we specify `T` via a member `prim` of `T`:

```
const value: unknown = {};
```

```
if (isTypeof(value, 123)) {
  // %inferred-type: number
  value;
}
```

23.3.2.2 Using overloading

A better solution is to use overloading (several cases are omitted):

```
/**
 * A partial implementation of the `typeof` operator.
 */
function isTypeof(value: any, typeString: 'boolean'): value is boolean;
function isTypeof(value: any, typeString: 'number'): value is number;
function isTypeof(value: any, typeString: 'string'): value is string;
function isTypeof(value: any, typeString: string): boolean {
  return typeof value === typeString;
}

const value: unknown = {};
if (isTypeof(value, 'boolean')) {
  // %inferred-type: boolean
  value;
}
```

(This approach is an idea by Nick Fisher.)

23.3.2.3 Using an interface as a type map

An alternative is to use an interface as a map from strings to types (several cases are omitted):

```
interface TypeMap {
  boolean: boolean;
  number: number;
  string: string;
}

/**
 * A partial implementation of the `typeof` operator.
 */
function isTypeof<T extends keyof TypeMap>(value: any, typeString: T)
: value is TypeMap[T] {
  return typeof value === typeString;
}

const value: unknown = {};
if (isTypeof(value, 'string')) {
  // %inferred-type: string
```

```
    value;
  }
```

(This approach is an idea by Ran Lottem.)

23.4 Assertion functions

An assertion function checks if its parameter fulfills certain criteria and throws an exception if it doesn't. For example, one assertion function supported by many languages, is `assert()`. `assert(cond)` throws an exception if the boolean condition `cond` is `false`.

On Node.js, `assert()` is supported via the built-in module `assert`. The following code uses it in line A:

```
  import assert from 'assert';
  function removeFilenameExtension(filename: string) {
    const dotIndex = filename.lastIndexOf('.');
    assert(dotIndex >= 0); // (A)
    return filename.slice(0, dotIndex);
  }
```

23.4.1 TypeScript's support for assertion functions

TypeScript's type inference provides special support for assertion functions, if we mark such functions with *assertion signatures* as return types. W.r.t. how and what we can return from a function, an assertion signature is equivalent to `void`. However, it additionally triggers narrowing.

There are two kinds of assertion signatures:

- Asserting a boolean argument: `asserts «cond»`
- Asserting the type of an argument: `asserts «arg» is «type»`

23.4.2 Asserting a boolean argument: `asserts «cond»`

In the following example, the assertion signature `asserts condition` states that the parameter `condition` must be `true`. Otherwise, an exception is thrown.

```
  function assertTrue(condition: boolean): asserts condition {
    if (!condition) {
      throw new Error();
    }
  }
```

This is how `assertTrue()` causes narrowing:

```
  function func(value: unknown) {
    assertTrue(value instanceof Set);

    // %inferred-type: Set<any>
    value;
  }
```

We are using the argument `value instanceof Set` similarly to a type guard, but instead of skipping part of a conditional statement, `false` triggers an exception.

23.4.3 Asserting the type of an argument: `asserts «arg» is «type»`

In the following example, the assertion signature `asserts value is number` states that the parameter `value` must have the type `number`. Otherwise, an exception is thrown.

```
function assertIsNumber(value: any): asserts value is number {
  if (typeof value !== 'number') {
    throw new TypeError();
  }
}
```

This time, calling the assertion function, narrows the type of its argument:

```
function func(value: unknown) {
  assertIsNumber(value);

  // %inferred-type: number
  value;
}
```

23.4.3.1 Example assertion function: adding properties to an object

The function `addXY()` adds properties to existing objects and updates their types accordingly:

```
function addXY<T>(obj: T, x: number, y: number)
: asserts obj is (T & { x: number, y: number }) {
  // Adding properties via = would be more complicated...
  Object.assign(obj, {x, y});
}

const obj = { color: 'green' };
addXY(obj, 9, 4);

// %inferred-type: { color: string; } & { x: number; y: number; }
obj;
```

An intersection type `S & T` has the properties of both type `S` and type `T`.

23.5 Quick reference: user-defined type guards and assertion functions

23.5.1 User-defined type guards

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

- Type predicate: `value is string`
- Result: `boolean`

23.5.2 Assertion functions

23.5.2.1 Assertion signature: **`asserts «cond»`**

```
function assertTrue(condition: boolean): asserts condition {
  if (!condition) {
    throw new Error(); // assertion error
  }
}
```

- Assertion signature: `asserts condition`
- Result: `void`, exception

23.5.2.2 Assertion signature: **`asserts «arg» is «type»`**

```
function assertIsString(value: unknown): asserts value is string {
  if (typeof value !== 'string') {
    throw new Error(); // assertion error
  }
}
```

- Assertion signature: `asserts value is string`
- Result: `void`, exception

23.6 Alternatives to assertion functions

23.6.1 Technique: forced conversion

An assertion function narrows the type of an existing value. A forced conversion function returns an existing value with a new type – for example:

```
function forceNumber(value: unknown): number {
  if (typeof value !== 'number') {
    throw new TypeError();
  }
  return value;
}

const value1a: unknown = 123;
// %inferred-type: number
const value1b = forceNumber(value1a);

const value2: unknown = 'abc';
assert.throws(() => forceNumber(value2));
```

The corresponding assertion function looks as follows:

```
function assertIsNumber(value: unknown): asserts value is number {
  if (typeof value !== 'number') {
    throw new TypeError();
  }
}

const value1: unknown = 123;
assertIsNumber(value1);
// %inferred-type: number
value1;

const value2: unknown = 'abc';
assert.throws(() => assertIsNumber(value2));
```

Forced conversion is a versatile technique with uses beyond those of assertion functions. For example, we can convert:

- From an input format (think JSON schema) that is easy to write
- Into an output format that is easy to work with in code.

For more information, see §24.3.5 "External vs. internal representation of data".

23.6.2 Technique: throwing an exception

Consider the following code:

```
function getLengthOfValue(strMap: Map<string, string>, key: string)
: number {
  if (strMap.has(key)) {
    const value = strMap.get(key);

    // %inferred-type: string | undefined
    value; // before type check

    // We know that value can't be `undefined`
    if (value === undefined) { // (A)
      throw new Error();
    }

    // %inferred-type: string
    value; // after type check

    return value.length;
  }
  return -1;
}
```

Instead of the `if` statement that starts in line A, we also could have used an assertion function:

```
assertNotUndefined(value);
```

Throwing an exception is a quick alternative if we don't want to write such a function. Similarly to calling an assertion function, this technique also updates the static type.

23.7 `@hqoss/guards`: library with type guards

The library `@hqoss/guards` provides a collection of type guards for TypeScript – for example:

- Primitives: `isBoolean()`, `isNumber()`, etc.
- Specific types: `isObject()`, `isNull()`, `isFunction()`, etc.
- Various checks: `isNonEmptyArray()`, `isInteger()`, etc.

Chapter 24

Validating external data

Contents

Data validation means ensuring that data has the desired structure and content.

With TypeScript, validation becomes relevant when we receive external data such as:

- Data parsed from JSON files
- Data received from web services

In these cases, we expect the data to fit static types we have, but we can't be sure. Contrast that with data we create ourselves, where TypeScript continuously checks that everything is correct.

This chapter explains how to validate external data in TypeScript.

24.1 JSON schema

Before we can explore approaches for data validation in TypeScript, we need to take a look at *JSON schema* because several of the approaches are based on it.

The idea behind JSON schema is to express the *schema* (structure and content, think static type) of JSON data in JSON. That is, metadata is expressed in the same format as data.

The use cases for JSON schema are:

- Validating JSON data: If we have a schema definition for data, we can use tools to check that the data is correct. One issue with data can also be fixed automatically: We can specify default values that can be used to add properties that are missing.
- Documenting JSON data formats: On one hand, the core schema definitions can be considered documentation. But JSON schema additionally supports descriptions, deprecation notes, comments, examples, and more. These mechanisms are called *annotations*. They are not used for validation, but for documentation.
- IDE support for editing data: For example, Visual Studio Code supports JSON schema. If there is a schema for a JSON file, we gain several editing features: auto-completion, highlighting of errors, etc. Notably, VS Code's support for `package.json` files is completely based on a JSON schema.

24.1.1 An example JSON schema

This example is taken from the `json-schema.org` website:

```
{
  "$id": "https://example.com/geographical-location.schema.json",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Longitude and Latitude Values",
  "description": "A geographical coordinate.",
  "required": [ "latitude", "longitude" ],
  "type": "object",
  "properties": {
    "latitude": {
      "type": "number",
      "minimum": -90,
      "maximum": 90
    },
    "longitude": {
      "type": "number",
      "minimum": -180,
      "maximum": 180
    }
  }
}
```

The following JSON data is valid w.r.t. this schema:

```
{
  "latitude": 48.858093,
  "longitude": 2.294694
}
```

24.2 Approaches for data validation in TypeScript

This section provides a brief overview of various approaches for validating data in TypeScript. For each approach, I list one or more libraries that support the approach. W.r.t. libraries, I don't intend to be comprehensive because things change quickly in this space.

24.2.1 Approaches not using JSON schema

- Approach: Invoking builder methods and functions to produce both validation functions (at runtime) and static types (at compile time). Libraries include:
  - `github.com/vriad/zod` (demonstrated later in this chapter)
  - `github.com/pelotom/runtypes`
  - `github.com/gcanti/io-ts`
- Approach: Invoking builder methods and only producing validation functions. Libraries taking this approach often focus on making validation as versatile as possible:
  - `github.com/sindresorhus/ow`
  - `github.com/hapijs/joi`
  - `github.com/jquense/yup`
- Approach: Compiling TypeScript types to validation code at compile time. Libraries:
  - TypeScript transformer: `github.com/ts-type-makeup/superstruct-ts-transformer`
  - Babel macro: `github.com/vedantroy/typecheck.macro`

24.2.2 Approaches using JSON schema

- Approach: Converting TypeScript types to JSON schema. Libraries:
  - `github.com/vega/ts-json-schema-generator`
  - `github.com/YousefED/typescript-json-schema`
- Approach: Converting a JSON schema to TypeScript types. Libraries:
  - `github.com/quicktype/quicktype`
  - `github.com/bcherny/json-schema-to-typescript`
  - `github.com/Q42/json-typescript-decoder`
- Approach: Validating JSON data via JSON schemas. This functionality is also useful for the previous two approaches. npm packages:
  - `github.com/ajv-validator/ajv`
  - `github.com/geraintluff/tv4`

24.2.3 Picking a library

Which approach and therefore library to use, depends on what we need:

- If we are starting with TypeScript types and want to ensure that data (coming from configuration files, etc.) fits those types, then builder APIs that support static types are a good choice.

- If our starting point is a JSON schema, then we should consider one of the libraries that support JSON schemas.

- If we are handling data that is more messy (e.g. submitted via forms), we may need a more flexible approach where static types play less of a role.

24.3 Example: validating data via the library *Zod*

24.3.1 Defining a "schema" via Zod's builder API

Zod has a builder API that produces both types and validation functions. That API is used as follows:

```
import * as z from 'zod';

const FileEntryInputSchema = z.union([
  z.string(),
  z.tuple([z.string(), z.string(), z.array(z.string())]),
  z.object({
    file: z.string(),
    author: z.string().optional(),
    tags: z.array(z.string()).optional(),
  }),
]);
```

For larger schemas, it can make sense to break things up into multiple `const` declarations.

Zod can produce a static type from `FileEntryInputSchema`, but I decided to (redundantly!) manually maintain the static type `FileEntryInput`:

```
type FileEntryInput =
  | string
  | [string, string, string[]]
  | {file: string, author?: string, tags?: string[]}
  ;
```

Why the redundancy?

- It's easier to read.
- It helps with migrating to a different validation library or approach, should I ever have to.

Zod's generated type is still helpful because we can check if it's assignable to `FileEntryInput`. That will warn us about most problems related to the two getting out of sync.

24.3.2 Validating data

The following function checks if the parameter `data` conforms to `FileEntryInputSchema`:

```
function validateData(data: unknown): FileEntryInput {
  return FileEntryInputSchema.parse(data); // may throw an exception
}

validateData(['iceland.txt', 'me', ['vacation', 'family']]); // OK
```

```
assert.throws(
  () => validateData(['iceland.txt', 'me']));
```

The static type of the result of `FileEntryInputSchema.parse()` is what Zod derived from `FileEntryInputSchema`. By making `FileEntryInput` the return type of `validateData()`, we ensure that the former type is assignable to the latter.

24.3.3 Type guards

`FileEntryInputSchema.check()` is a type guard:

```
function func(data: unknown) {
  if (FileEntryInputSchema.check(data)) {
    // %inferred-type: string
    // | [string, string, string[]]
    // | {
    //     author?: string | undefined;
    //     tags?: string[] | undefined;
    //     file: string;
    //   }
    data;
  }
}
```

It can make sense to define a custom type guard that supports `FileEntryInput` instead of what Zod infers.

```
function isValidData(data: unknown): data is FileEntryInput {
  return FileEntryInputSchema.check(data);
}
```

24.3.4 Deriving a static type from a Zod schema

The parameterized type `z.infer<Schema>` can be used to derive a type from a schema:

```
// %inferred-type: string
// | [string, string, string[]]
// | {
//     author?: string | undefined;
//     tags?: string[] | undefined;
//     file: string;
//   }
type FileEntryInputDerived = z.infer<typeof FileEntryInputSchema>;
```

24.3.5 External vs. internal representation of data

When working with external data, it's often useful to distinguish two types.

On one hand, there is the type that describes the input data. Its structure is optimized for being easy to author:

```
type FileEntryInput =
  | string
  | [string, string, string[]]
  | {file: string, author?: string, tags?: string[]}
  ;
```

On the other hand, there is the type that is used in the program. Its structure is optimized for being easy to use in code:

```
type FileEntry = {
  file: string,
  author: null|string,
  tags: string[],
};
```

After we have used Zod to ensure that the input data conforms to `FileEntryInput`, we use a conversion function that converts the data to a value of type `FileEntry`.

24.4 Conclusion

In the long run, I expect compile-time solutions that derive validation functions from static types, to improve w.r.t. usability. That should make them more popular.

For libraries that have builder APIs, I'd like to have tools that compile TypeScript types to builder API invocations (online and via a command line). This would help in two ways:

- The tools can be used to explore how the APIs work.
- We have the option of producing API code via the tools.