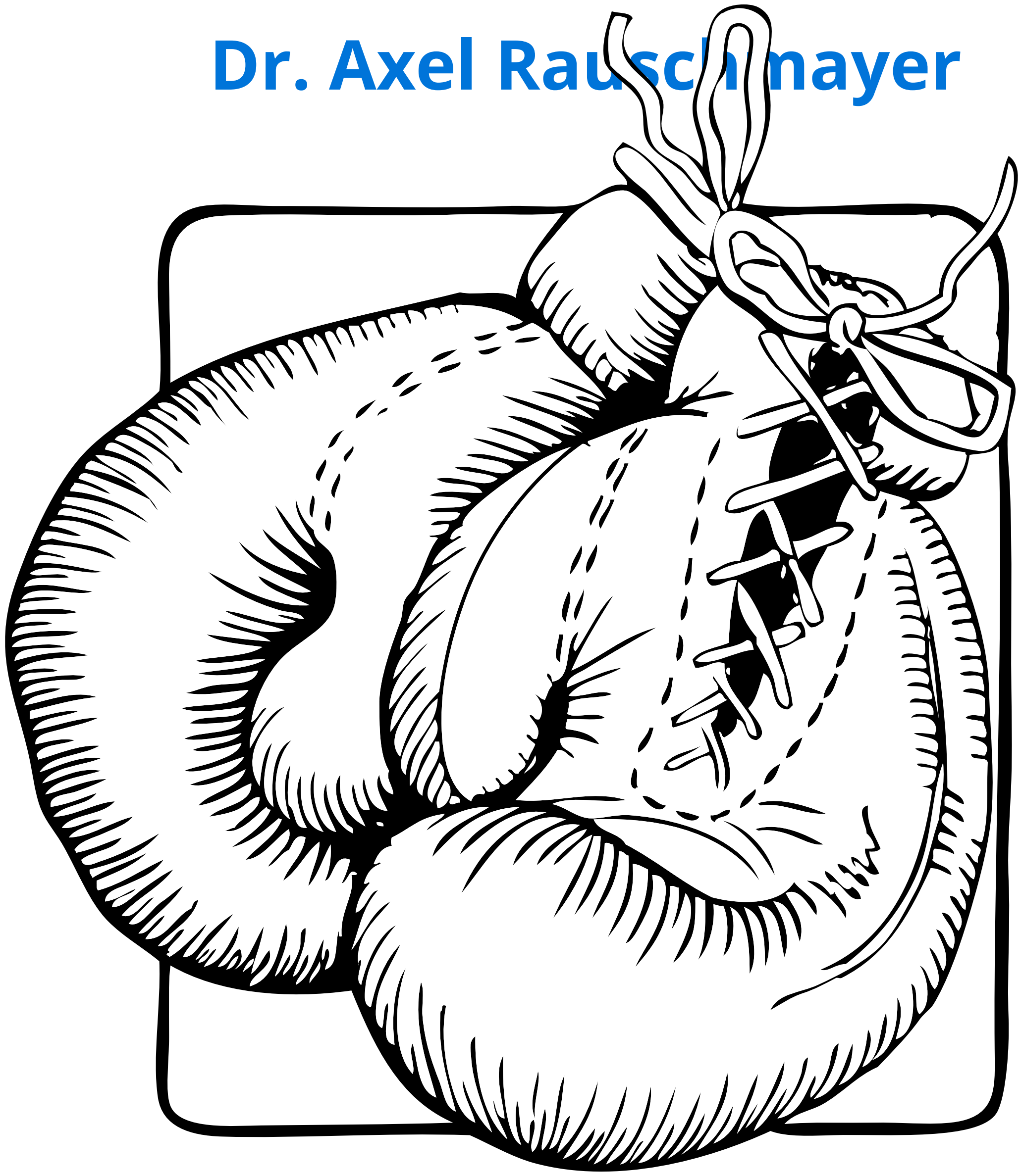
Dr. Axel Rauschmayer



Tackling TypeScript

Upgrading from JavaScript

Tackling TypeScript

Dr. Axel Rauschmayer

2020

Copyright © 2020 by Dr. Axel Rauschmayer

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

exploringjs.com

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 `toString()`, 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 [content not included]. 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
  ```

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

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

Naming type parameters

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

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 generic class

Classes can have type parameters, too:

```
class SimpleStack<Elem> {
  #data: Array<Elem> = [];
  push(x: Elem): void {
    this.#data.push(x);
  }
  pop(): Elem {
    const result = this.#data.pop();
    if (result === undefined) {
        throw new Error();
    }
    return result;
  }
  get length() {
    return this.#data.length;
  }
}
```

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

```
const stringStack = new SimpleStack<string>();
stringStack.push('first');
stringStack.push('second');
```

```
assert.equal(stringStack.length, 2);
assert.equal(stringStack.pop(), 'second');
```

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 and methods

Function definitions can introduce type variables like this:

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

We use the function as follows.

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

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

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

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

7.15.2.1 Arrow functions and methods

Arrow functions can also have type parameters:

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

This is the type parameter syntax for methods:

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

7.15.3 A more complicated function example

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

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

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

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

```
// %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:

```
{
  "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:

```
<!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

Basic types

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 Two questions for each perspective

The following two questions are important for understanding how types work and need to be answered from each of the two perspectives.

1. What does it mean for `myVariable` to have the type `MyType`?

   ```
   let myVariable: MyType;
   ```

2. How is `UnionType` derived from `Type1`, `Type2`, and `Type3`?

   ```
   type UnionType = 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`, then all values that can be assigned to `myVariable` must be elements of the set `MyType`.

2. 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.
- If a location whose type is `Src` is assigned to a location whose type is `Trg`, TypeScript determines if the assignment is legal via the type relationship *assignment compatibility*. Rules include:
  - `Src` is assignable to `Trg` if `Src` and `Trg` are identical types.
  - `Src` is assignable to `Trg` if `Src` or `Trg` is the type `any`.
  - `Src` is assignable to `Trg` if `Src` is a string literal type and `Trg` is the primitive type `string`.
  - `Src` is assignable to `Trg` if `Src` is a union type and each constituent type of `Src` is assignable to `Trg`.
  - `Src` is assignable to `Trg` if `Trg` is a union type and `Src` is assignable to at least one constituent type of `Trg`.
  - Etc.

Let's consider the questions:

1. If `myVariable` has the type `MyType`, then we can only assign values to it whose static types are assignment-compatible with `MyType`.
2. How union types work is also determined via assignment compatibility. We have seen two relevant rules.

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

Where are the remaining chapters?

You are reading a preview of this book:

- The full version of this book is available for purchase.
- You can take a look at the full table of contents (also linked to from the book's homepage).