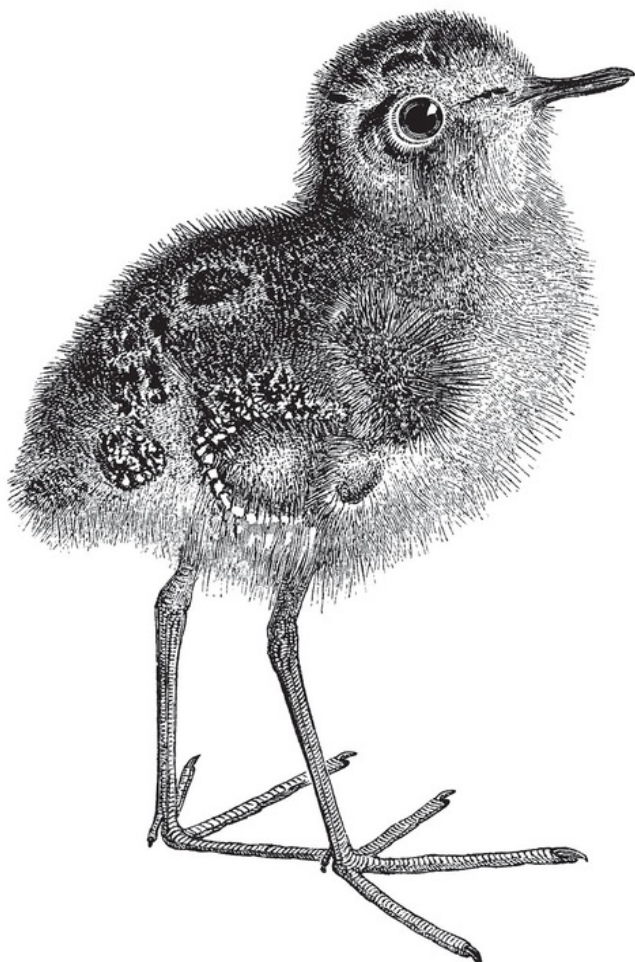


O'REILLY®

TypeScript Cookbook

Solutions for Everyday Problems



Early
Release

RAW &
UNEDITED

Stefan Baumgartner

TypeScript Cookbook

Solutions for Everyday Problems

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Stefan Baumgartner



Beijing • Boston • Farnham • Sebastopol • Tokyo

TypeScript Cookbook

by Stefan Baumgartner

Copyright © 2023 Stefan Baumgartner. All rights reserved.

Printed in the United States of America.

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

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

- Acquisitions Editor: Amanda Quinn
- Development Editor: Shira Evans
- Production Editor: Elizabeth Faerm
- Copyeditor:
- Proofreader:
- Indexer:
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- October 2023: First Edition

Revision History for the Early Release

- 2022-09-14: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098136659> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *TypeScript Cookbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have

used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-13659-8

[LSI]

Chapter 1. Project Setup

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

You want to get started with TypeScript, fantastic! The big question is: How do you start? There are many ways how you can integrate TypeScript into your projects, and they all are slightly different depending on your project’s needs. Just as JavaScript runs on many runtimes, there are plenty of ways to configure TypeScript so it meets your target’s needs.

This chapter covers all the possibilities of introducing TypeScript to your project, as an extension next to JavaScript that gives you basic auto-completion and error indication, up to full-fledge setups for full-stack applications on Node.js and the browser.

Since JavaScript tooling is a field with endless possibilities — some say that a new JavaScript build chain is released every week, almost as much as new frameworks — this chapter focuses more on things you can do with the TypeScript compiler alone and without any extra tool.

TypeScript offers everything you need for your transpilation needs, except the ability to create minified and optimized bundles for web distribution. Bundlers like **ESBuild** or **Webpack** take care of this task. Also, there are setups that include other transpilers like **Babel.js** which can play along nicely with TypeScript. Bundlers and other transpilers are not in the scope of this chapter. Refer to their documentation for the inclusion of TypeScript, and use the knowledge in this chapter to get the right configuration setup.

With TypeScript being a project with more than a decade worth of history, it carries some remains from older times, that for the sake of compatibility TypeScript can't just get rid of. Therefore, this chapter will spotlight modern JavaScript syntax and recent developments in web standards. If you still need to target Internet Explorer 8 or Node.js 10, then first: I'm sorry, these platforms are really hard to develop for. And second: You will be able to put together the pieces for older platforms with the knowledge from this chapter and the [official TypeScript documentation](#).

1.1 Type-checking JavaScript

Problem

You want to get basic type-checking for JavaScript with the lowest amount of effort possible.

Solution

Add a single-line comment with `@ts-check` at the beginning of every JavaScript file you want to type-check. With the right editors, you already get red squiggly lines whenever TypeScript encounters things that don't quite add up.

Discussion

TypeScript has been designed as a superset of JavaScript, and every valid JavaScript is also valid TypeScript. This means that TypeScript is also really good at figuring out potential errors in regular JavaScript code.

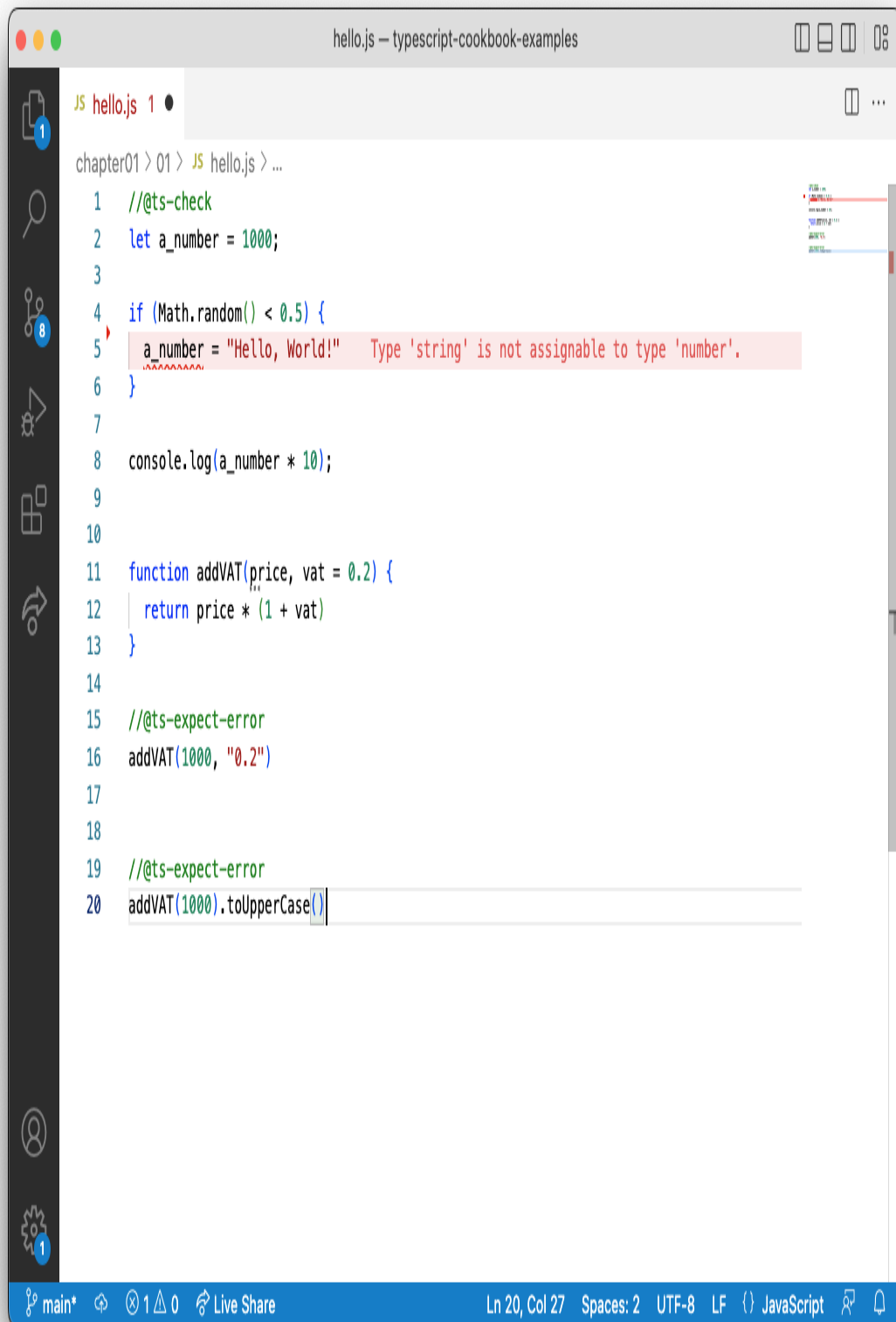
We can make use of this if we don't want to have a full-blown TypeScript setup but want some basic hints and type-checks to ease our development workflow.

A good pre-requisite if you only want to type-check JavaScript is a good editor or integrated development environment (IDE). An editor that goes really well with TypeScript is [Visual Studio Code](#). Visual Studio Code — or

VSCode for short — was the first major project to utilize TypeScript, even far before TypeScript's release.

A lot of people recommend VSCode if you want to write JavaScript or TypeScript. But really, every editor is great as long as it features TypeScript support. And nowadays most of them do.

With Visual Studio Code we get one very important thing for type-checking JavaScript: Red squiggly lines when something doesn't quite add up, as you can see in **Figure 1-1**. This is probably the lowest barrier to entry.



```
hello.js — typescript-cookbook-examples

JS hello.js 1 •

chapter01 > 01 > JS hello.js > ...

1 //@ts-check
2 let a_number = 1000;
3
4 if (Math.random() < 0.5) {
5   a_number = "Hello, World!" Type 'string' is not assignable to type 'number'.
6 }
7
8 console.log(a_number * 10);
9
10
11 function addVAT(price, vat = 0.2) {
12   return price * (1 + vat)
13 }
14
15 //@ts-expect-error
16 addVAT(1000, "0.2")
17
18
19 //@ts-expect-error
20 addVAT(1000).toUpperCase()
```

main* 1 0 Live Share Ln 20, Col 27 Spaces: 2 UTF-8 LF {} JavaScript

Figure 1-1. Red squiggly lines in code editors: First-level feedback if something in your code doesn't add up.

TypeScript's type system has different levels of strictness when working with a codebase.

First, the type system will try to *infer* types from JavaScript code through usage. If you have a line like this in your code:

```
let a_number = 1000;
```

TypeScript will correctly infer `number` as the type of `a_number`.

One difficulty with JavaScript is that types are dynamic. Bindings via `let`, `var`, or `const` can change type based on usage. Take a look at the following example:

```
let a_number = 1000;

if (Math.random() < 0.5) {
  a_number = "Hello, World!"
}

console.log(a_number * 10);
```

We assign a number to `a_number` and change the binding to a `string` if the condition in the next line evaluates to true. This wouldn't be that much of a problem if we wouldn't try to multiply `a_number` on the last line. In approximately 50% of all cases, this example will produce unwanted behavior.

TypeScript can help us here. With the addition of a single-line comment with `@ts-check` at the very top of our JavaScript file, TypeScript activates the next strictness level: Type-checking JavaScript files based on the type information available in the JavaScript file.

In our example, TypeScript will figure out that we try to assign a string to a binding that TypeScript has inferred to be a number. We will get an error in

our editor.

```
// @ts-check
let a_number = 1000;

if (Math.random() < 0.5) {
  a_number = "Hello, World!"
// ^-- Type 'string' is not assignable to type 'number'.ts(2322)
}

console.log(a_number * 10);
```

Now we can start to fix our code, TypeScript will guide us.

Type inference for JavaScript goes a long way. In the following example, TypeScript infers types by looking at operations like multiplication and addition, as well as default values.

```
function addVAT(price, vat = 0.2) {
  return price * (1 + vat)
}
```

The function `addVAT` takes two arguments. The second argument is optional, as it has been set to a default value of `0.2`. TypeScript will tell you if you try to pass a value that doesn't work.

```
addVAT(1000, "a string")
//           ^-- Argument of type 'string' is not assignable
//           to parameter of type 'number'.ts(2345)
```

Also, since we use multiplication and addition operations within the function body, TypeScript understands that we will return a number from this function.

```
addVAT(1000).toUpperCase()
//           ^-- Property 'toUpperCase' does not
//           exist on type 'number'.ts(2339)
```

There are situations where you need more than type-inference. In JavaScript files, you can annotate function arguments and bindings through JSDoc

type annotations. TypeScript will pick up your annotations and uses them as types for the type system.

```
/** @type {number} */
let amount;

amount = '12';
//      ^-- Argument of type 'string' is not assignable
//           to parameter of type 'number'.ts(2345)

/**
 * Adds VAT to a price
 *
 * @param {number} price The price without VAT
 * @param {number} vat The VAT [0-1]
 *
 * @returns {number}
 */
function addVAT(price, vat = 0.2) {
  return price * (1 + vat)
}
```

JSDoc also allows you to define new, complex types for objects.

```
/**
 * @typedef {Object} Article
 * @property {number} price
 * @property {number} vat
 * @property {string} string
 * @property {boolean=} sold
 */

/**
 * Now we can use Article as a proper type
 * @param {[Article]} articles
 */
function totalAmount(articles) {
  return articles.reduce((total, article) => {
    return total + addVAT(article)
  }, 0)
}
```

The syntax might feel a bit clunky, though, we will find better ways to annotate objects in [Recipe 1.3](#).

Given that you have a JavaScript codebase that is well documented via JSDoc, adding a single line on top of or files will give you already a really good understanding if something goes wrong in your code.

1.2 Installing TypeScript

Problem

Red squiggles in the editor are not enough, you want to have command line feedback, status codes, configuration, and options to type-check JavaScript and compile TypeScript.

Solution

Install TypeScript via Node's primary package repository: **NPM**

Discussion

TypeScript is written in TypeScript, compiled to JavaScript, and uses the **Node.js JavaScript runtime** as its primary execution environment.¹. So even if you're not writing a Node.js app, the tooling for your JavaScript applications will run on Node.

So, make sure you get Node.js from **the official website** and get familiar with its command line tools.

For a new project, make sure you initialize your project's folder with a fresh **package.json**. This file contains all the information for Node and its package manager NPM to figure out your project's contents. Generate a new **package.json** file with default contents in your project's folder with the NPM command line tool.

```
$ npm init -y
```

NOTE

Throughout this book, we will see commands that should be executed in your terminal. For the sake of convenience, we decided to show these commands as they would appear on BASH or similar shells that are available for Linux, macOS, or the Windows subsystem for Linux. The leading \$ sign is a convention to indicate a command but is not meant to write by you. Note that all commands also work on the regular Windows command line interface as well as PowerShell.

npm is Node's package manager. It comes with a CLI, a registry, and other tools that allow you to install dependencies. Once you initialized your `package.json`, install TypeScript from NPM. We install it as a development dependency, meaning that TypeScript won't be included if you intend to publish your project as a library to NPM itself.

```
$ npm install -D typescript
```

There is a way to globally install TypeScript so you have the TypeScript compiler available everywhere, but I'd strongly suggest installing TypeScript separately per project. Depending on how frequently you visit your projects, you will end up with different TypeScript versions that are in sync with your project's code. Installing (and updating) TypeScript globally might break projects you haven't touched in a while.

NOTE

If you install front-end dependencies via NPM, you will need an additional tool to make sure that your code also runs in your browser: A bundler. TypeScript doesn't include a bundler that works with the supported module systems, so you need to make sure that you have the proper tooling set up. Tools like **Webpack** are common, and so is **ESBuild**. All tools are designed to execute TypeScript as well. Or you can go full native, as described in **Recipe 1.8**

Now that TypeScript is installed, initialize a new TypeScript project. Use NPX for that, it allows you to execute a command line utility that you installed relatively to your project.

With

```
$ npm tsc --init
```

you can run your project's local version of the TypeScript compiler, and pass the `"init"` flag to create a new `tsconfig.json`.

The `tsconfig.json` is the main configuration file for your TypeScript project. It contains all the configuration needed TypeScript understands how to interpret your code, how to make types available for dependencies, and if you need to turn certain features on or off.

Per default, TypeScript sets these options for you:

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}
```

Let's look at them in detail.

`target` is `es2016`, which means that if you run the TypeScript compiler, it will compile your TypeScript files to an ECMAScript 2016 compatible syntax. Depending on your supported browsers or environments, you can either set that to something more recent (ECMAScript versions are nowadays named after the year of release), or something almost ancient. Like `es5` for people who have to support very old Internet Explorer versions. We all hope you don't have to.

`module` is `commonjs`. This allows you to write ECMAScript module syntax, but instead of carrying this syntax over to the output, TypeScript will compile it to the CommonJS format. This means that a

```
import { name } from "../my-module"
```

```
console.log(name)
//...
```

becomes a

```
const my_module_1 = require("./my-module");
console.log(my_module_1.name);
```

once you compile. CommonJS was the module system for Node.js and has become very common because of Node's popularity. Node.js since has adopted ECMAScript modules as well, something that we tackle in [Recipe 1.9](#).

`esModuleInterop` makes sure that modules that aren't ECMAScript modules are aligned to the standard once imported.

`forceConsistentCasingInFileNames` helps people who have case-sensitive file systems cooperating with folks who have case insensitive file systems. And `skipLibCheck` assumes that your installed type definition files (more on that later) have no errors. So your compiler won't check them and will become a little faster.

One of the most interesting features is TypeScript's strict mode. If set to true, TypeScript will behave differently in certain areas. It's the way for the TypeScript team to define their view on how the type system should behave.

If TypeScript introduces a breaking change because their view on the type system changes, it will get incorporated in strict mode. This ultimately means that your code might break if you update TypeScript and always run in strict mode.

To give you time to adapt to changes, TypeScript also allows you to turn certain strict mode features on or off feature by feature.

Additionally to the default settings, I strongly recommend setting two more:

```
{
  "compilerOptions": {
    //...
    "rootDir": "./src",
```

```
    "outDir": "./dist"  
  }  
}
```

You tell TypeScript to pick up source files from an `src` folder and put the compiled files into a `dist` folder. This setup allows you to separate your built files from the ones you author. You will have to create the `src` folder, of course, the `dist` folder will be created after you compile.

Oh, compilation. Once you have your project set up, create an `index.ts` file in `src`.

```
console.log("Hello World")
```

The `.ts` extension indicates it's a TypeScript file. Now run

```
$ npx tsc
```

in your command line and see the compiler at work.

1.3 Keeping types on the side

Problem

You want to write regular JavaScript with no extra build-step, but still get some editor support and proper type information for your functions. But you don't want to define your complex object types with JSDoc like shown in [Recipe 1.1](#).

Solution

Keep type definition files “on the side” and run the TypeScript compiler in the “check JavaScript” mode.

Discussion

Gradual adoption has always been a dedicated goal for TypeScript. With this technique, which I dubbed “types on the side”, you are able to write TypeScript syntax for object types and advanced features like generics and conditional types (see Chapter 5) instead of clunky JSDoc comments, but still write JavaScript for your actual app.

Somewhere in your project, maybe in a `@types` folder, create a type definition file. Its ending is `.d.ts`, and as opposed to regular `.ts` files, its purpose is to hold declarations, no actual code.

This is where you can write your interfaces, type aliases, and complex types.

```
// @types/person.d.ts

// An interface for objects of this shape
export interface Person {
  name: string,
  age: number
}

// An interface that extends the original one
// this is tough to write with JSDoc comments alone.
export interface Student extends Person {
  semester: number
}
```

Note that you export the interfaces from the declaration files. This is so you can import them in your JavaScript files.

```
// index.js
/** @typedef { import ("../@types/person").Person } Person */
```

The comment on the very first line tells TypeScript to import the `Person` type from `@types/person`, and make it available under the name `Person`.

Now you can use this identifier to annotate function parameters or objects just like you would with primitive types like `string`.

```
// index.js, continued

/**
 * @param {Person} person
 */
function printPerson(person) {
  console.log(person.name);
}
```

To make sure that you get editor feedback, you still need to set `@ts-check` at the beginning of your JavaScript files as described in [Recipe 1.1](#). Or, you can configure your project to always check JavaScript.

Open `tsconfig.json` and set the `checkJs` flag to `true`. This will pick up all the JavaScript files from your `src` folder and give you constant feedback on type errors in your editor. You are also able to run `npx tsc` to see if you have errors in your command line.

If you don't want TypeScript to transpile your JavaScript files to older versions of JavaScript, make sure you set `noEmit` to `true`.

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

With that, TypeScript will have a look at your source files, and will give you all the type information you need, but won't touch your code.

This technique is also known to scale. Prominent JavaScript libraries like [Preact](#) work like this and are able to provide fantastic tooling for their users as well as their contributors.

1.4 Migrating a project to TypeScript

Problem

You want to get the full benefits of TypeScript for your project, but you need to migrate an entire code-base.

Solution

Rename your modules file by file from `*.js` to `*.ts`. Make use of several compiler options and features that help you iron out errors.

Discussion

The benefit of having TypeScript files instead of JavaScript files with types is that you get to have your types and implementations in one file, which gives you better editor support, access to more TypeScript features, and increases compatibility with other tools.

However, just renaming all files from `.js` to `.ts` will result most likely in one thing: Tons and tons of errors. This is why you should go file by file, and gradually increase type safety as you go along.

The biggest problem when migrating is that you're suddenly dealing with a TypeScript project, not with JavaScript anymore. But still, lots of your modules will be JavaScript, and with no type information, they will fail the type checking step.

Make it easier for you and for TypeScript if you turn off type checking for JavaScript, but allow for TypeScript modules to load and refer to JavaScript files.

```
{
  "compilerOptions": {
    "checkJs": false,
    "allowJS": true
  }
}
```

Should you run `npx tsc` now, you will see that TypeScript picks up all JavaScript and TypeScript files in your source folder and creates respective JavaScript files in your destination folder. TypeScript will also transpile your code to be compatible with the specified target version.

If you are working with dependencies a lot, you will see that some of them don't come with type information. This will also produce TypeScript errors.

```
import _ from "lodash"
//           ^- Could not find a declaration
//           file for module 'lodash'.
```

Install third-party type definitions to get rid of this error. See [Recipe 1.5](#)

Once you migrate file by file, you might realize that you won't be able to get all typings for one file right in one go. There are dependencies, and you will quickly go down the rabbit hole of having too many files to adjust before you can tackle the one that you actually need.

You can always decide to just live with the error. By default, TypeScript sets the compiler option `noEmitOnError` to false.

```
{
  "compilerOptions": {
    "noEmitOnError": false
  }
}
```

This means that no matter how many errors you have in your project, TypeScript will generate result files, trying not to block you. This might be a setting you want to turn on after you finished migrating.

In strict mode, TypeScript's feature flag `noImplicitAny` is set to true. This flag will make sure that you don't forget to assign a type to a variable, constant, or function parameter. Even if it's just any.

```
function printPerson(person: any) {
  // This doesn't make sense, but is ok with any
  console.log(person.gobbledegook)
}

// This also doesn't make sense, but any allows it
printPerson(123)
```

Any is the catch-all type in TypeScript. Every value is compatible with any, and any allows you to access every property or call every method. Any

effectively turns off type-checking, giving you some room to breathe during your migration process.

Alternatively, you can annotate your parameters with `unknown`. This also allows you to pass everything to a function, but won't allow you to do anything with it until you make sure you know more about the type.

You can also decide to ignore errors by adding a `@ts-ignore` comment before the line you want to exclude from type checking. A `@ts-nocheck` comment at the beginning of your file turns off type checking entirely for this particular module.

A comment directive that is fantastic for migration is `@ts-expect-error`. It works like `@ts-ignore` as it will swallow errors from the type-checking progress, but will produce red squiggly lines if there is no type error to be found.

```
function printPerson(person: Person) {  
    console.log(person.name)  
}  
  
// This error will be swallowed  
// @ts-expect-error  
printPerson(123)  
  
function printNumber(nr: number) {  
    console.log(nr);  
}  
  
// v- Unused '@ts-expect-error' directive.ts(2578)  
// @ts-expect-error  
printNumber(123)
```

The great thing about this technique is that you flip responsibilities.

Usually, you would have to make sure that you pass in the right values to a function, now you can make sure that the function is able to handle the right input.

All possibilities to get rid of errors throughout your migration process have one thing in common: They're explicit. You need to explicitly set `@ts-expect-error` comments, annotate function parameters as `any`, or

ignore files entirely from type checking. With that, you can always search for those escape hatches during the migration process, and make sure that over time, you got entirely rid of them.

1.5 Loading types from Definitely Typed

Problem

You rely on a dependency that hasn't been written in TypeScript and therefore lacks typings.

Solution

Install community-maintained type definitions from [Definitely Typed](#).

Discussion

Definitely Typed is one of the biggest and most active repositories on GitHub, and collects high-quality TypeScript type definitions developed and maintained by the community.

The number of maintained type definitions is close to 10.000, and there is rarely a JavaScript library **not** available.

All type definitions are linted, checked, and deployed to the Node.js package registry NPM under the `@types` namespace. NPM has an indicator on each package's information site that shows if Definitely Typed type definitions are available, as you can see in [Figure 1-2](#).

react - npm

npmjs.com/package/react

Nightmare Prom Memories

ProductsPricingDocumentation

npm

Search packages

Search

Sign Up

Sign In

react

18.2.0 • Public • Published a month ago

ReadmeExplore BETA1 Dependency0 Dependents978 Versions

react

React is a JavaScript library for creating user interfaces.

The `react` package contains only the functionality necessary to define React components. It is typically used together with a React renderer like `react-dom` for the web, or `react-native` for the native environments.

Note: by default, React will be in development mode. The development version includes extra warnings about common mistakes, whereas the production version includes extra performance optimizations and strips all error messages. Don't forget to use the **production build** when deploying your application.

Usage

```
import { useState } from 'react';
import { createRoot } from 'react-dom/client';

function Counter() {
  const [count, setCount] = useState(0);
```

Install

> npm i react

Repository

github.com/facebook/react

Homepage

reactjs.org/

Weekly Downloads

15,757,161

Version	License
18.2.0	MIT

Unpacked Size	Total Files
316 kB	20

IssuesPull Requests

Figure 1-2. The NPM site for React shows a DT logo right next to the package name. This indicates available type definitions from Definitely Typed.

A click on this logo leads you to the actual site for type definitions. If a package has first-party type definitions already available, it shows a small TS logo right next to the package name, as shown in **Figure 1-3**.

@types/react - npm

npmjs.com/package/@types/react

Nighttime Possum Meandering

ProductsPricingDocumentation

npm

Search packages

Search

Sign Up

Sign In

@types/react

TS

18.0.15 • Public • Published 16 days ago

Readme

Explore

BETA

3 Dependencies

12,051 Dependents

401 Versions

Installation

Install

npm install --save @types/react

> npm i @types/react

Summary

Repository

github.com/DefinitelyTyped/Def...

Homepage

github.com/DefinitelyTyped/Def...

Weekly Downloads

16,514,706

Version

18.0.15

License

MIT

Unpacked Size

175 kB

Total Files

10

Issues

Pull Requests

Details

This package contains type definitions for React

(http://facebook.github.io/react/).

Files were exported from

https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types/react.

Additional Details

Last updated: Tue, 05 Jul 2022 23:02:57 GMT

Dependencies: @types/csstype, @types/prop-types, @types/scheduler

Global values: React



Figure 1-3. Type Definitions for React from Definitely Typed

To install e.g. typings for the popular JavaScript framework React, you install the `@types/react` package to your local dependencies.

```
# Installing React
$ npm install --save react

# Installing Type Definitions
$ npm install --save @types/react
```

By default, TypeScript will pick up any type definitions it can find that are in visible `@types` folders relative to your project's root folder. It will also pick up all type definitions from `node_modules/@types`, this is where NPM installs e.g. `@types/react`.

This is because the `typeRoots` compiler option in `tsconfig.json` is set to `@types` and `./node_modules/@types`. Should you have the need to override this setting, please make sure to include the original folders if you still want to pick up type definitions from Definitely Typed.

```
{
  "compilerOptions": {
    "typeRoots": [". typings", ". node_modules/@types"]
  }
}
```

Note that by just installing type definitions into `node_modules/@types`, TypeScript will load them during compilation. This means that if some types declare globals, TypeScript will pick them up.

You might want to explicitly specify which packages should be allowed to contribute to the global scope by specifying them in the `types` setting in your compiler options.

```
{
  "compilerOptions": {
```

```
    "types": ["node", "jest"]
  }
}
```

Note that this setting will only affect the contributions to the global scope. If you load node modules via import statements, TypeScript still will pick up the correct types from `@types`:

```
// If `@types/lodash` is installed, we get proper  
// type definitions for this NPM package  
import _ from "lodash"  
  
const result = _.flattenDeep([1, [2, [3, [4]], 5]]);
```

We will revisit this setting in [Recipe 1.7](#).

1.6 Setting up a full-stack project

Problem

You want to write a full-stack application targeting Node.js and the browser, with shared dependencies.

Solution

Create two `tsconfig` files for each front-end and back-end, and load shared dependencies as composites.

Discussion

Node.js and the browser both run JavaScript, but they have a very different understanding of what developers should do with the environment. Node.js is meant for servers, command line tools, and everything that runs without a UI — *headless*. It has its own set of APIs and standard library. This little script starts an HTTP server.

```

const http = require('http'); // 1

const hostname = '127.0.0.1';
const port = process.env.PORT || 3000; // 2

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
  // 3
});

```

While it's without a doubt JavaScript, there are some things unique to Node.js.

1. "http" is a built-in Node.js module for everything related to HTTP. It is loaded via `require`, which is an indicator for Node's module system called *CommonJS*. There are other ways to load modules in Node.js nowadays as we see in [Recipe 1.9](#), but CommonJS has been the most common as of recent.
2. The `process` object is a global object containing information on environment variables and the current Node.js process in general. This is also unique to Node.js
3. The `console` and its functions are available in almost every JavaScript runtime, but what it does in Node is different from what it does in the browser. In Node, it prints on STDOUT, in the browser, it will print a line to the development tools.

There are of course many more unique APIs for Node.js. But the same goes for JavaScript in the browser.

```

import { msg } from `./msg.js`; // 1

document.querySelector('button')?.addEventListener("click", () => {
  // 2
  console.log(msg); // 3
});

```

1. After years without a way to load modules, ECMAScript modules have found their way into JavaScript and the browsers. This line loads an object from another JavaScript module. This runs in the browser natively, and is a second module system for Node.js (see [Recipe 1.9](#)).
2. JavaScript in the browser is meant to interact with UI events. The `document` object and the idea of a `querySelector` that points to elements in the DOM are unique to the browser. So is adding an event listener and listening on “click” events. You don’t have this in Node.js.
3. And again, `console`. It has the same API as in Node.js, but the result is a bit different.

The differences are so big, that it’s hard to create one TypeScript project that handles both. If you are writing a full-stack application, you need to create two TypeScript configuration files that deal with each part of your stack.

Let’s work on the back-end first. Let’s assume you want to write an Express.js server in Node.js (Express is a popular server framework for Node). First, you create a new NPM project as shown in [Recipe 1.1](#). Then, install Express as a dependency

```
$ npm install --save express
```

And install type definitions for Node.js and Express from Definitely Typed.

```
$ npm install -D @types/express @types/node
```

Create a new folder called “server”. This is where your Node.js code goes. Instead of creating a new `tsconfig.json` via `tsc`, create a new `tsconfig.json` in your project’s “server” folder. This is the contents:

```
// server/tsocnfig.json
{
  "compilerOptions": {
    "target": "ESNext",
    "lib": ["ESNext"],
```

```

    "module": "commonjs",
    "rootDir": "./",
    "moduleResolution": "node",
    "types": ["node"],
    "outDir": "../dist/server",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}

```

You should already know a lot of things, but a few things stand out:

- The `module` property is set to `commonjs`, the original Node.js module system. All `import` and `export` statements will be transpiled to their CommonJS counterpart
- The `types` property is set to `["node"]`. This property includes all the libraries you want to have globally available. If `"node"` is in the global scope, you will get type information for `require`, `process`, and other Node.js specifics that are in the global space.

To compile your server-side code, run

```
$ npx tsc -p server/tsconfig.json
```

Now for the client.

```

// client/tsconfig.json
{
  "compilerOptions": {
    "target": "ESNext",
    "lib": ["DOM", "ESNext"],
    "module": "ESNext",
    "rootDir": "./",
    "moduleResolution": "node",
    "types": [],
    "outDir": "../dist/client",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}

```

There are a few similarities, but again, some things stand out.

1. You add “DOM” to the `lib` property. This gives you type definitions for everything related to the browser. Where you needed to install Node.js typings via Definitely Typed, TypeScript ships the most recent type definitions for the browser with the compiler.
2. The `types` array is empty. This will *remove* “node” from our global typings. Since you only can install type definitions per `package.json`, the “node” type definitions we installed earlier would be available in the entire code base. For the `client` part though, you want to get rid of them.

To compile your front-end code, run

```
$ npx tsc -p client/tsconfig.json
```

Please note that you configured two distinct `tsconfig.json` files. Editors like *Visual Studio Code* pick up configuration information only for `tsconfig.json` files per folder. You could as well name them `tsconfig.server.json` and `tsconfig.client.json` and have them in your project’s root folder (and adjust all directory properties). `tsc` will use the correct configurations and throw errors if it finds some, but the editor will mostly stay silent or work with a default configuration.

Things get a bit hairier if you want to have shared dependencies. One way to achieve shared dependencies is to use project references and composite projects. This means that you extract your shared code in its own folder, but tell TypeScript that this is meant to be a dependency project of another one.

Create a `shared` folder on the same level as `client` and `server`.

Create a `tsconfig.json` in `shared` with the following contents.

```
// shared/tsconfig.json
{
  "compilerOptions": {
    "composite": true,
    "target": "ESNext",
    "module": "ESNext",
    "rootDir": "../shared/",
  }
}
```

```

    "moduleResolution": "Node",
    "types": [],
    "declaration": true,
    "outDir": "../dist/shared",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  },
}

```

Two things stand out again:

1. The flag `composite` is set to `true`. This allows other projects to reference this one.
2. The `declaration` flag is also set to `true`. This will generate `d.ts` files from your code so other projects can consume type information.

To include them in your client and server code, add this line to `client/tsconfig.json` and `server/tsconfig.json`.

```

// server/tsconfig.json
// client/tsconfig.json
{
  "compilerOptions": {
    // Same as before
  },
  "references": [
    { "path": "../shared/tsconfig.json" }
  ]
}

```

And you are all set. You can write shared dependencies and include them in your client and server code.

There is a caveat, however. This works great if you only share e.g. models and type information, but the moment you share actual functionality, you will see that the two different module systems (CommonJS in Node, ECMAScript modules in the browser) can't be unified in one compiled file. You either create an ESM module and can't import it in CommonJS code, or you create CommonJS code and can't import it in the browser.

There are two things you can do:

1. Compile to CommonJS and let a bundler take care of the module resolution work for the browser.
2. Compile to ECMAScript modules and write modern Node.js applications based on ECMAScript modules. See [Recipe 1.9](#) for more information.

Since you start out new, I would strongly recommend the second option.

1.7 Setting up tests

Problem

You want to write tests, but the globals for testing frameworks interfere with your production code.

Solution

Create a separate `tsconfig` for development and build, and exclude all test files in the latter one.

Discussion

In the JavaScript and Node.js ecosystem, there are a lot of unit testing frameworks and test runners. They vary in detail, have different opinions, or are tailored for certain needs. Some of them you might just find prettier than others.

While test runners like [Ava](#) rely on importing modules to get the framework into scope, others provide a set of globals. Take [Mocha](#) for example.

```
import assert from "assert";
import { add } from "..";

describe("Adding numbers", () => {
  it("should add two numbers", () => {
    assert.equal(add(2, 3), 5);
  })
})
```

`assert` comes from the Node.js built-in assertion library, but `describe`, `it` and many more are globals provided by Mocha. They also only exist when the Mocha CLI is running.

This provides a bit of a challenge for your type setup, as those functions are necessary to write tests, but aren't available when you execute your actual application.

The solution is to create two different configuration files. A regular `tsconfig.json` for development that your editor can pick up (remember [Recipe 1.6](#)), and a separate `tsconfig.build.json` that you use when you want to compile your application.

The first one includes all the globals you need, including types for Mocha, the latter one makes sure no test file is included within your compilation.

Let's go through this step by step. We look at Mocha as an example, but other test runners which provide globals (like [Jest](#)) work just the same way.

First, install Mocha and its types.

```
$ npm install --save-dev mocha @types/mocha @types/node
```

Create a new `tsconfig.base.json`. Since the only differences between development and build are the set of files to be included and the libraries activated, you want to have all the other compiler settings located in one file you can reuse for both. An example file for a Node.js application would look like this:

```
// tsconfig.base.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "commonjs",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "outDir": "./dist",
    "skipLibCheck": true
  }
}
```

The source files should be located `src`, test files should be located in an adjacent folder `test`. The setup you create in this recipe will also allow you to create files ending with `.test.ts` anywhere in your project.

Create a new `tsconfig.json` with your base development configuration. This one is used for editor feedback and for running tests with Mocha. You extend the basic settings from `tsconfig.base.json`, and inform TypeScript which folders to pick up for compilation.

```
// tsconfig.json
{
  "extends": "./tsconfig.base.json",
  "compilerOptions": {
    "types": ["node", "mocha"],
    "rootDirs": ["test", "src"]
  }
}
```

Note that you add `types` for Node and Mocha. The `types` property defines which globals are available, and in the development setting, you have both.

Additionally, you might find compiling your tests before executing them cumbersome. There are shortcuts to help you. For example, `ts-node` runs your local installation of Node.js and does an in-memory TypeScript compilation first.

```
$ npm install --save-dev ts-node
$ npx mocha -r ts-node/register tests/*.ts
```

With the development environment set up, it's time for the build environment. Create a `tsconfig.build.json`. It looks similar to `tsconfig.json`, but you will spot the difference right away.

```
// tsconfig.build.json
{
  "extends": "./tsconfig.base.json",
  "compilerOptions": {
    "types": ["node"],
    "rootDirs": ["src"]
  },
}
```

```
  "exclude": ["**/*.test.ts", "**/test/**"]
}
```

Additionally to changing `types` and `rootDirs`, you also define which files to exclude from type checking and compilation. You use wild-card patterns that exclude all files ending with `.test.ts` and are located in `test` folders. Depending on your taste, you can also add `.spec.ts` or `spec` folders to this array.

Compile your project by referring to the right JSON file.

```
$ npx tsc -p tsconfig.build.json
```

You will see that in the result files (located in `dist`), you won't see any test file. Also, while you still can access `describe` and `it` when editing your source files, you will get an error if you try to compile.

```
$ npx tsc -p tsconfig.build.json
```

```
src/index.ts:5:1 - error TS2593: Cannot find name 'describe'.
Do you need to install type definitions for a test runner?
Try `npm i --save-dev @types/jest` or `npm i --save-dev
@types/mocha`
and then add 'jest' or 'mocha' to the types field in your
tsconfig.
```

```
5 describe("this does not work", () => {})
   ~~~~~~
```

```
Found 1 error in src/index.ts:5
```

If you don't like polluting your globals during development mode, you can choose a similar setup as in [Recipe 1.6](#), but it won't allow you to write tests adjacent to your source files.

Additionally, you can always opt for a test runner that prefers the module system.

1.8 Typing ECMAScript modules from URLs

Problem

You want to work without bundlers and use the browser's module loading capabilities for your app, yet you still want to have all the type information.

Solution

Set `target` and `module` in your `tsconfig`'s compiler options to `esnext`, point to your modules with a `.js` extension. Additionally, install types to dependencies via NPM, and use the `path` property in your `tsconfig` to tell TypeScript where to look for types.

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
    "paths": {
      "https://esm.sh/lodash@4.17.21": [
        "node_modules/@types/lodash/index.d.ts"
      ]
    }
  }
}
```

Discussion

Modern browser support module loading out of the box. Instead of bundling your app into a smaller set of files, you can use the raw JavaScript files directly.

CDNs² like esm.sh, [UnPKG](https://unpkg.com), and others are designed to distribute node modules and JavaScript dependencies as URLs, consumable by native ECMAScript module loading.

With proper caching and state-of-the-art HTTP, ECMAScript modules become a real alternative for apps.

TypeScript has no modern bundler included, so you would need to install an extra tool anyway. But if you decide to go module first, there are a few things to consider when working with TypeScript.

What you want to achieve is to write `import` and `export` statements in TypeScript, but preserve the module loading syntax, and let the browser handle module resolution.

```
// File module.ts
export const obj = {
  name: 'Stefan'
}

// File index.ts
import { obj } from './module'

console.log(obj.name)
```

To achieve this, tell TypeScript to

1. Compile to an ECMAScript version that understands modules
2. Use the ECMAScript module syntax for module code generation

Update two properties in your `tsconfig.json`.

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
  }
}
```

`module` tells TypeScript how to transform import and export statements. The default converts module loading to CommonJS, as seen in [XREF HERE](#). Setting `module` to `esnext` will use ECMAScript module loading, and thus preserve the syntax.

`target` tells TypeScript the ECMAScript version you want to transpile your code to. Once a year, there's a new ECMAScript release with new features. Setting `target` to `esnext` will always target the latest ECMAScript version.

Depending on your compatibility goals, you might want to set this property to the ECMAScript version compatible with the browsers you want to

support. This is usually a version with a year (e.g. es2015, es2016, es2017, etc). ECMAScript modules work with every version from es2015 onwards. If you go for an older version, you won't be able to load ECMAScript modules natively in the browser.

Changing these compiler options already does one important thing: It leaves the syntax intact. A problem occurs once you want to run our code.

Usually, import statements in TypeScript point to files without an extension. You write `import { obj } from './module'`, leaving out `.ts`. Once you compile, this extension is still missing. But the browser needs an extension to actually point to the respective JavaScript file.

The solution: Add a `.js` extension, even though you are pointing to a `.ts` file when you develop. TypeScript is smart enough to pick that up.

```
// index.ts

// This still loads types from 'module.ts', but keeps
// the reference intact once we compile it.
import { obj } from './module.js'

console.log(obj.name)
```

For your project's modules, that's all you need!

It gets a lot more interesting when you want to use dependencies. If you go native, you might want to load modules from a CDN, like esm.sh.

```
import _ from "https://esm.sh/lodash@4.17.21"
//                               ^- Error 2307

const result = _.flattenDeep([1, [2, [3, [4]], 5]]);

console.log(result)
```

TypeScript will error with the following message: *Cannot find module ... or its corresponding type declarations.(2307)*

TypeScript's module resolution works when files are on your disk, not on some server via HTTP. To get the info we need, we have to provide

TypeScript with a resolution of our own.

Even though we are loading dependencies from URLs, the type information for these dependencies lives with NPM. For lodash, you can install type information from Definitely Typed:

```
$ npm install -D @types/lodash
```

For dependencies that come with their own types, you can install the dependencies directly.

```
$ npm install -D preact
```

Once the types are installed, use the `paths` property in your compiler options to tell TypeScript how to resolve your URL.

```
// tsconfig.json
{
  "compilerOptions": {
    // ...
    "paths": {
      "https://esm.sh/lodash@4.17.21": [
        "node_modules/@types/lodash/index.d.ts"
      ]
    }
  }
}
```

Be sure to point to the right file!

There's also an escape hatch if you don't want to use typings, or if you just can't find typings. Within TypeScript, we can use `any` to intentionally disable type-checking. For modules, we can do something very similar: We can ignore the TypeScript error.

```
// @ts-ignore
import _ from "https://esm.sh/lodash@4.17.21"
```

`ts-ignore` removes the **next** line from type checking and can be used everywhere, where you want to ignore type errors (See XREF HERE). This

effectively means that you won't get any type information for your dependencies and you might run into errors, but it might be the ultimate solution for unmaintained, old dependencies that you just need, but won't find any types for.

1.9 Loading different module types in Node

Problem

You want to make use of ECMAScript modules in Node.js and the CommonJS interoperability feature for libraries.

Solution

Set TypeScript's module resolution to "nodeNext" and name your files .mts or .cts.

Discussion

With the advent of Node.js, the CommonJS module system has become one of the most popular module systems in the JavaScript ecosystem and has continued to be popular for a very long time.

The idea is simple and effective: Define exports in one module, and require them in another.

```
// person.js
function printPerson(person) {
  console.log(person.name);
}

exports = {
  printPerson
}

// index.js
const person = require('./person');
person.printPerson({ name: "Stefan", age: 40});
```

This system has been a huge influence on ECMAScript modules and has also been the default for TypeScript's module resolution and transpiler. This means that with the `commonjs` module setting, your `import` and `export` statements are transpiled to `require` and `exports`.

Example 1-1. Code example

```
// person.ts
type Person = {
  name: string,
  age: number
}

export function printPerson(person) {
  console.log(person.name);
}

// index.ts
import * as person from "./person";
person.printPerson({ name: "Stefan", age: 40});
```

With ECMAScript modules stabilizing, also Node.js has started to adopt them. Even though the basics of both module systems seem to be very similar, there are some differences in the details, like handling default exports, or the fact that you can load ECMAScript modules asynchronously.

As there is no way to treat both module systems the same but with different syntax, the Node.js maintainers decided to give both systems room, and assigned different file endings to indicate the preferred module type.

Table 1-1 shows the different endings, how they're named in TypeScript, what TypeScript compiles them to, and what they can import. Thanks to the CommonJS interoperability, it's fine to import CommonJS modules from ECMAScript modules, but not the other way around.

Table 1-1. Module endings and what they import

Ending	TypeScript	Compiles to	Can import
.js	.ts	CommonJS	.js, .cjs
.cjs	.cts	CommonJS	.js, .cjs

Ending TypeScript Compiles to Can import			
.mjs	.mts	ES Modules	.js, .cjs, .mjs

Library developers who publish on NPM get extra information in their `package.json` file to indicate the main type of a package (`module` or `commonjs`), and to point to a list of main files or fallbacks so module loaders can pick up the right file.

```
// package.json
{
  "name": "dependency",
  "type": "module",
  "exports": {
    ".": {
      // Entry-point for `import "dependency"` in ES Modules
      "import": "./esm/index.js",
      // Entry-point for `require("dependency")` in CommonJS
      "require": "./commonjs/index.cjs",
    },
  },
  // CommonJS Fallback
  "main": "./commonjs/index.cjs",
}
```

In TypeScript, you mainly write ECMAScript module syntax and let the compiler decide which module format to create in the end. Now there are possibly two of them: CommonJS and ECMAScript modules.

To allow for both, you can set module resolution in your `tsconfig.json` to `NodeNext`.

```
{
  "compilerOptions": {
    "module": "NodeNext"
    // ...
  }
}
```

With that flag, TypeScript will pick up the right the modules as described in your dependencies `package.json`, will recognize `mts` and `cts` endings, and will follow [Table 1-1](#) for module imports.

For you as a developer, there are differences in importing files. Since CommonJS didn't require endings when importing, TypeScript still supports imports without endings. The example in [Example 1-1](#) still works, if all you use is CommonJS.

Importing with file endings, just like in [Recipe 1.8](#), allow modules to be imported in both ECMAScript modules and CommonJS modules.

```
// index.mts
import * as person from "./person.js"; // works in both
person.printPerson({ name: "Stefan", age: 40});
```

Should CommonJS interoperability not work, you can always fall back on a `require` statement. Add "node" as global types to your compiler options.

```
// tsconfig.json
{
  "compilerOptions": {
    "module": "NodeNext",
    "types": ["node"],
  }
}
```

Then, import with this TypeScript-specific syntax:

```
// index.mts
import person = require("./person.cjs");

person.printPerson({ name: "Stefan", age: 40 });
```

In a CommonJS module, this will be just another `require` call, in ECMAScript modules, this will include Node.js helper functions.

```
// compiled index.mts
import { createRequire as _createRequire } from "module";
const __require = _createRequire(import.meta.url);
const person = __require("./person.cjs");
person.printPerson({ name: "Stefan", age: 40 });
```

Note that this will reduce compatibility to non-Node.js environments like the browser, but might eventually fix interoperability issues.

1.10 Working with Deno and dependencies

Problem

You want to use TypeScript with Deno, a modern JavaScript runtime for applications outside the browser.

Solution

That's relatively easy, TypeScript is built-in.

Discussion

Deno is a modern JavaScript runtime created by the same people who developed Node.js in the first place. Deno is very similar to Node.js in many ways, but where it isn't the differences are significant:

- Deno adopts web platform standards for their main APIs, meaning that you will have it easier to port code from the browser to the server.
- It bets heavily on security, only allowing file system or network access if you explicitly activate it.
- Dependencies are not handled via a centralized registry, but — again adopting browser features — via URLs.

Oh, and it comes with built-in development tooling and TypeScript!

Deno is ultimately the tool with the lowest barrier if you want to try out TypeScript. No need to download any other tool (the `tsc` compiler is already built-in), no need for TypeScript configurations. You write `.ts` files and Deno handles the rest.

```
// main.ts
function sayHello(name: string) {
  console.log(`Hello ${name}`)
```

```
}  
  
sayHello("Stefan")  
  
$ deno run main.ts
```

Deno's TypeScript can do everything `tsc` can do, and is updated with every Deno update. However, there are some differences when you want to configure it.

First, the default configuration is already slightly different in its default settings as opposed to the default configuration issued by `tsc --init`. Strict mode feature flags are set differently, and there's support for React (on the server-side!) already included.

To make changes to the configuration, you should create a `deno.json` file in your roots folder. This will be picked up by Deno automatically unless you tell it not to. `deno.json` includes several configurations for the Deno runtime, including TypeScript compiler options.

```
{  
  "compilerOptions": {  
    // Your TSC compiler options  
  },  
  "fmt": {  
    // Options for the auto-formatter  
  },  
  "lint": {  
    // Options for the linter  
  }  
}
```

You can see more on what's possible on the [Deno website](#).

Also, the default libraries are different. Even though Deno supports web platform standards and has browser-compatible APIs, it needs to make some cuts because of the fact that there is no graphical user interface. That's why some types e.g. the "DOM" library just clash with what can be provided by Deno.

Some libraries of interest are:

- `deno.ns`, the default Deno namespace.
- `deno.window`, the global object for Deno.
- `deno.worker`, the equivalent for Web Workers in the Deno runtime.

“DOM” and subsets still are included in Deno, but not switched on by default. If your application targets both the browser and Deno, configure Deno to include all browser and Deno libraries:

```
// deno.json
{
  "compilerOptions": {
    "target": "esnext",
    "lib": ["dom", "dom.iterable", "dom.asynciterable",
    "deno.ns"]
  }
}
```

An example of a framework that targets both Deno and the browser is [Aleph.js](#).

One thing that’s also different with Deno is how type information for dependencies is distributed. External dependencies in Deno are loaded via URLs from a CDN. Deno itself hosts its standard library at <https://deno.land/std>.

But you can also use CDNs like [esm.sh](#) or [UnPKG](#), like in [Recipe 1.8](#). All these CDNs distribute types by sending an `X-TypeScript-Types` header with the HTTP request, showing Deno was to load type declarations. This also goes for dependencies that don’t have first-party type declarations but rely on Definitely Typed.

So the moment you install your dependency, Deno will fetch not only the source files but also all the type information.

If you don’t load a dependency from a CDN, but rather have it locally, you can point to a type declaration file the moment you import the dependency

```
// @deno-types="./charting.d.ts"
import * as charting from "./charting.js";
```

or include a reference to the typings in the library itself

```
// charting.js  
/// <reference types="./charting.d.ts" />
```

This reference is also called a triple-slash directive and a TypeScript feature, not a Deno feature. There are various triple-slash directives, mostly used for pre-ECMAScript module dependency systems. The [docs](#) gives a really good overview. If you stick with ECMAScript modules, you most likely won't use them, though.

1.11 Using pre-defined configurations

Problem

You want to use TypeScript for a certain framework or platform, but don't know where to start with your configuration.

Solution

Use a pre-defined configuration from [tsconfig/bases](#) and extend from.

Discussion

Just like Definitely Typed hosts community-maintained type definitions for popular libraries, [tsconfig/bases](#) hosts a set of community-maintained recommendations for TypeScript configurations you can use as a starting point for your own project. This includes frameworks like Ember.js, Svelte, or Next.js, but also JavaScript runtimes like Node.js and Deno.

The configuration files are reduced to a minimum, mostly dealing with recommended libraries, module, and target settings, and a bunch of strict mode flags that make sense for the respective environment.

For example, this is the recommended configuration for Node.js 18, with a recommended strict mode setting, and with ECMAScript modules:


```
{
  "$schema": "https://json.schemastore.org/tsconfig",
  "display": "Node 18 + ESM + Strictest",
  "compilerOptions": {
    "lib": [
      "es2022"
    ],
    "module": "es2022",
    "target": "es2022",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "moduleResolution": "node",
    "allowUnusedLabels": false,
    "allowUnreachableCode": false,
    "exactOptionalPropertyTypes": true,
    "noFallthroughCasesInSwitch": true,
    "noImplicitOverride": true,
    "noImplicitReturns": true,
    "noPropertyAccessFromIndexSignature": true,
    "noUncheckedIndexedAccess": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "importsNotUsedAsValues": "error",
    "checkJs": true
  }
}
```

To make use of this configuration, install it via NPM:

```
$ npm install --save-dev @tsconfig/node18-strictest-esm
```

and wire it up in your own TypeScript configuration:

```
{
  "extends": "@tsconfig/node18-strictest-esm/tsconfig.json",
  "compilerOptions": {
    // ...
  }
}
```

This will pick up all the settings from the pre-defined configuration. You can now start setting your own properties like root and out directories, and so on.

-
- ¹ TypeScript also works in other JavaScript runtimes, such as Deno and the browser, but they are not intended as main targets.
 - ² Content Delivery Networks

Chapter 2. Basic Types

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Now that you are all set up, it’s time to write some TypeScript! Starting out should be easy, but you will very soon run into situations where you’re unsure if you’re doing the right thing. Should you use interfaces or type aliases? Should you annotate or let type inference do its magic? What about `any` and `unknown`, are they safe to use? Some people on the internet said you should never use them, so why are they part of TypeScript anyways?

All these questions will be answered in this chapter. We look at the basic types that make TypeScript and learn how an experienced TypeScript developer will make use of them. Use this as a foundation for the upcoming chapters, so you get a feeling of how the TypeScript compiler gets to its types, and how it interprets your annotations.

This is a lot about the interaction between your code, the editor, and the compiler. And about going up and down the type hierarchy, which we will see in [XREF HERE](#). If you are an experienced TypeScript developer, this chapter will give you the missing foundation. If you’re just starting out, you will learn the most important techniques to go really far!

2.1 Annotating effectively

Problem

Annotating types is cumbersome and boring.

Solution

Only annotate when you want to have your types checked.

Discussion

A type annotation is a way to explicitly tell which types to expect. You know, the stuff that was very prominent in other programming languages, where the verbosity of `StringBuilder stringBuilder = new StringBuilder()` makes sure that you're really, really dealing with a `StringBuilder`. The opposite is type inference, where TypeScript tries to figure out the type for you.

```
// Type inference
let aNumber = 2;
// aNumber: number

// Type annotation
let anotherNumber: number = 3;
// anotherNumber: number
```

Type annotations are also the most obvious and visible syntax difference between TypeScript and JavaScript.

When you start learning TypeScript, you might want to annotate everything to express the types you'd expect. This might feel like the obvious choice but you can also use annotations sparingly and let TypeScript figure out types for you.

A type annotation is a way for you to express where contracts have to be checked. If you add a type annotation to a variable declaration, you tell the compiler to check if types match during the assignment.

```
type Person = {
  name: string,
  age: number
}
```

```
}  
  
const me: Person = createPerson()
```

If `createPerson` returns something that isn't compatible with `Person`, TypeScript will throw an error. Do this if you really want to be sure that you're dealing with the right type here.

Also, from that moment on, `me` is of type `Person`, and TypeScript will treat it as a `Person`. If there are more properties in `me`, e.g. a `profession`, TypeScript won't allow you to access them. It's not defined in `Person`.

If you add a type annotation to a function signature's return value, you tell the compiler to check if types match the moment you return that value.

```
function createPerson(): Person {  
    return { name: "Stefan", age: 39 }  
}
```

If I return something that doesn't match `Person`, TypeScript will throw an error. Do this if you want to be completely sure that you return the correct type. This especially comes in handy if you are working with functions that construct big objects from various sources.

If you add a type annotation to a function signature's parameters, you tell the compiler to check if types match the moment you pass along arguments.

```
function printPerson(person: Person) {  
    console.log(person.name, person.age)  
}  
  
printPerson(me)
```

This is in my opinion the most important, and unavoidable type annotation. Everything else can be inferred.

```
type Person = {  
    name: string,  
    age: number  
}
```

```

// Inferred!
// return type is { name: string, age: number }
function createPerson() {
    return { name: "Stefan", age: 39 }
}

// Inferred!
// me: { name: string, age: number}
const me = createPerson()

// Annotated! You have to check if types are compatible
function printPerson(person: Person) {
    console.log(person.name, person.age)
}

// All works
printPerson(me)

```

You can use inferred object types at places where you expect an annotation because TypeScript has a *structural type system*. In a structural type system, the compiler will only take into account the members (properties) of a type, not the actual name.

Types are compatible if all members of the type to check against are available in the type of the value. We also say that the *shape* or *structure* of a type has to match.

```

type Person = {
    name: string,
    age: number
}

type User = {
    name: string,
    age: number,
    id: number
}

function printPerson(person: Person) {
    console.log(person.name, person.age)
}

const user: User = {
    name: "Stefan",
    age: 40,
    id: 815
}

```

```
}  
  
printPerson(user) // works!
```

User has more properties than Person, but all properties that are in Person are also in User, and they have the same type. This is why it's possible to pass User objects to printPerson, even though the types don't have any explicit connection.

However, if you pass a literal, TypeScript will complain that there are excess properties that should not be there.

```
printPerson({  
  name: "Stefan",  
  age: 40,  
  id: 1000  
})  
// ^- Argument of type '{ name: string; age: number; id: number; }'  
//    is not assignable to parameter of type 'Person'.  
//    Object literal may only specify known properties,  
//    and 'id' does not exist in type 'Person'.(2345)
```

This is to make sure that you didn't expect properties to be present in this type, and wonder yourself why changing them has no effect.

With a structural type system, you can create interesting patterns where you have carrier variables with the type inferred, and reuse the same variable in different parts of your software, with no similar connection to each other.

```
type Person = {  
  name: string,  
  age: number  
}  
  
type Studying = {  
  semester: number  
}  
  
type Student = {  
  id: string,  
  age: number,  
  semester: number  
}
```

```

function createPerson() {
    return { name: "Stefan", age: 39, semester: 25, id: "XPA"}
}

function printPerson(person: Person) {
    console.log(person.name, person.age)
}

function studyForAnotherSemester(student: Studying) {
    student.semester++
}

function isLongTimeStudent(student: Student) {
    return student.age - student.semester / 2 > 30 &&
    student.semester > 20
}

const me = createPerson()

// All work!
printPerson(me)
studyForAnotherSemester(me)
isLongTimeStudent(me)

```

Student, Person, and Studying have some overlap, but are unrelated to each other. `createPerson` returns something that is compatible with all three types. If you have annotated too much, you would need to create a lot more types and a lot more checks than necessary, without any benefit.

So annotate wherever you want to have your types checked, but at least for function arguments.

2.2 Working with any and unknown

Problem

There are two top types in TypeScript, `any` and `unknown`. Which one should you use?

Solution

Use `any` if you effectively want to deactivate typing, use `unknown` when you need to be cautious.

Discussion

Both `any` and `unknown` are top types, which means that every value is compatible with `any` or `unknown`.

```
const name: any = "Stefan";
const person: any = { name: "Stefan", age: 40 };
const notAvailable: any = undefined;
```

Since `any` is a type every value is compatible with, you can access any property without restriction.

```
const name: any = "Stefan";
// This is ok for TypeScript, but will crash in JavaScript
console.log(name.profession.experience[0].level);
```

`any` is also compatible with every sub-type, except `never`. This means you can narrow the set of possible values by assigning a new type.

```
const me: any = "Stefan";
// Good!
const name: string = me;
// Bad, but ok for the type system.
const age: number = me;
```

With `any` being so permissive, `any` can be a constant source of potential errors and pitfalls since you effectively deactivate type checking.

While everybody seems to agree that you shouldn't use `any` in your codebases, there are some situations where `any` is really useful:

Migration. When you go from JavaScript to TypeScript, chances are that you already have a large codebase with a lot of implicit information on how your data structures and objects work. It might be a chore to get everything spelled out in one go. `any` can help you migrate to a safer codebase incrementally.

Untyped Third-party dependencies. You might have one or the other JavaScript dependency that still refuses to use TypeScript (or something similar). Or even worse: There are no up-to-date types for it. Definitely Typed is a great resource, but it's also maintained by volunteers. It's a formalization of something that exists in JavaScript but is not directly derived from it. There might be errors (even in such popular type definitions like React's), or they just might not be up to date!

This is where `any` can help you greatly. When you know how the library works, if the documentation is good enough to get you going, and if you use it sparingly, `any` can be a relief instead of fighting types.

JavaScript prototyping. TypeScript works a bit differently from JavaScript and needs to make a lot of trade-offs to make sure that you don't run into edge cases. This also means that if you write certain things that would work in JavaScript, you'd get errors in TypeScript.

```
type Person = {
  name: string,
  age: number
}

function printPerson(person: Person) {
  for(let key in person) {
    console.log(`${key}: ${person[key]}`)
    // Element implicitly has an 'any' --^
    // type because expression of type 'string'
    // can't be used to index type 'Person'.
    // No index signature with a parameter of type 'string'
    // was found on type 'Person'.(7053)
  }
}
```

Find out why this is an error in [XREF HERE](#). In cases like this, `any` can help you to switch off type checking for a moment because you know what you're doing. And since you can go from every type to `any`, but also back to every other type, you have little, explicit unsafe blocks throughout your code where you are in charge of what's happening.

```
function printPerson(person: any) {
  for(let key in person) {
```

```

    console.log(`${key}: ${person[key]}`)
  }
}

```

Once you know that this part of your code works, you can start adding the right types, work around TypeScript's restrictions, and type assertions.

```

function printPerson(person: Person) {
  for(let key in person) {
    console.log(`${key}: ${person[key as keyof Person]}`)
  }
}

```

Whenever you use `any`, make sure you activate the `noImplicitAny` flag in your `tsconfig.json`; it is activated by default in `strict` mode. With that, TypeScript needs you to explicitly annotate `any` when you don't have a type through inference or annotation. This helps find potentially problematic situations later on.

An alternative to `any` is `unknown`. It allows for the same values, but the things you can do with it are very different. Where `any` allows you to do everything, `unknown` allows you to do nothing. The only thing you can do is pass values around, the moment you want to call a function or make the type more specific, you need to do type checks first.

```

const me: unknown = "Stefan";
const name: string = me;
//    ^- Type 'unknown' is not assignable to type 'string'.(2322)
const age: number = me;
//    ^- Type 'unknown' is not assignable to type 'number'.(2322)

```

Type checks and control flow analysis help to do more with `unknown`:

```

function doSomething(value: unknown) {
  if(typeof value === "string") {
    // value: string
    console.log("It's a string", value.toUpperCase());
  } else if (typeof value === "number") {
    // value: number
    console.log("it's a number", value * 2)
  }
}

```

If your apps should work with a lot of different types, `unknown` is great to make sure that you can carry values throughout your code, but don't run into any safety problems because of `any`'s permissiveness.

2.3 Choosing the right object type

Problem

You want to allow for values that are JavaScript objects, but there are three different object types, `object`, `Object` and `{}`, which one should you use?

Solution

Use `object` for compound types like objects, functions, and arrays. `{}` for everything that has a value.

Discussion

TypeScript divides its types into two branches. The first branch, primitive types, include `number`, `boolean`, `string`, `symbol`, `bigint`, and some sub-types. The second branch is called *compound types* and includes everything that is a sub-type of an object and is ultimately composed of other compound types or primitive types. [Figure 2-1](#) gives an overview.

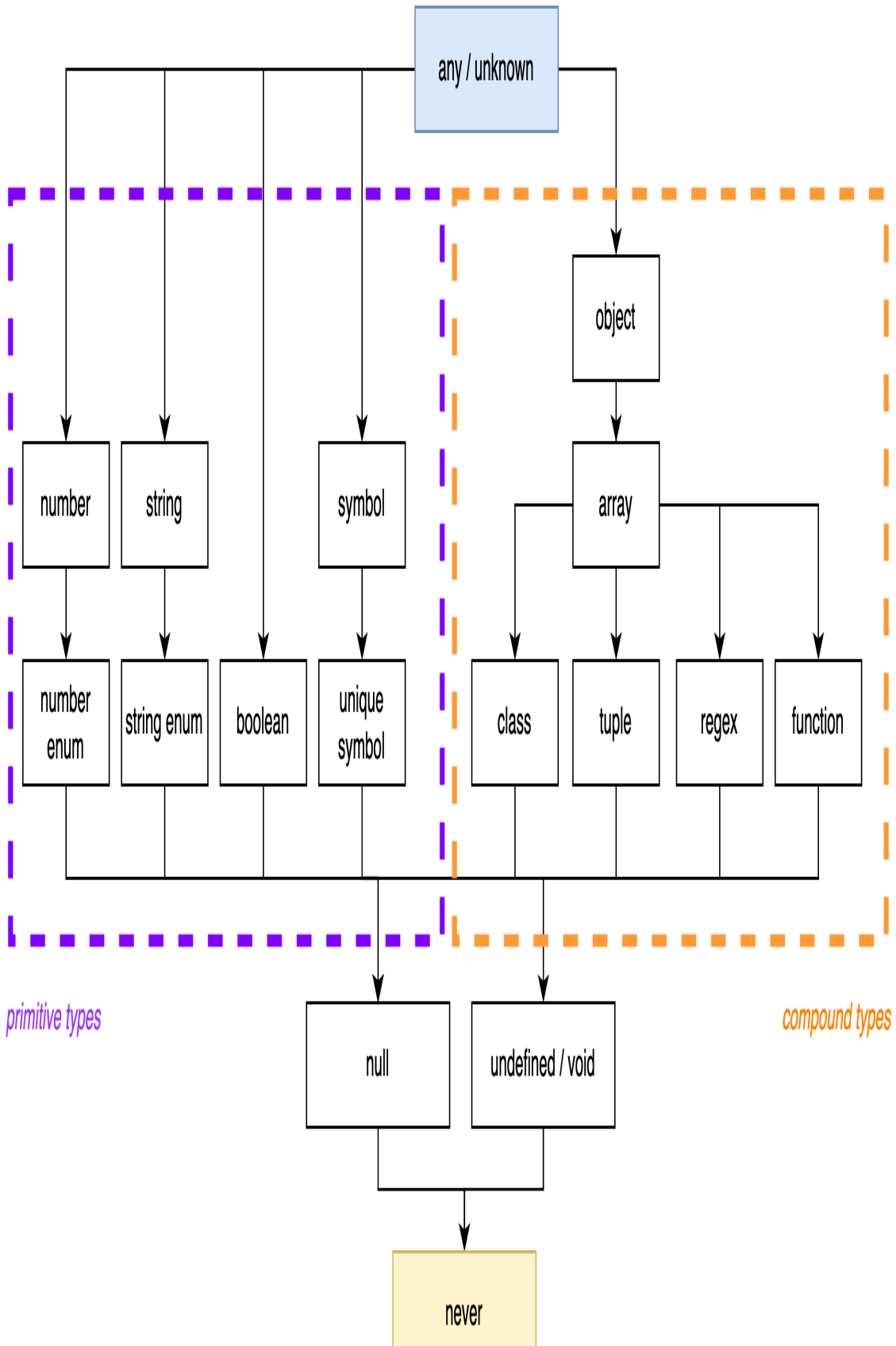




Figure 2-1. The type hierarchy in TypeScript.

There are situations where you want to target values that are *compound types*. Either because you want to modify certain properties, or just want to be safe that we don't pass any primitive values. For example `Object.create` creates a new object and takes its prototype as the first argument. This can only be a *compound type*, otherwise, your runtime JavaScript code would crash.

```
Object.create(2);  
// Uncaught TypeError: Object prototype may only be an Object or  
// null: 2  
// at Function.create (<anonymous>)
```

In TypeScript, there are three types that seem to do the same thing: The empty object type `{}`, the uppercase `Object` interface, and the lowercase `object` type. Which one do you use for compound types?

`{}` and `Object` allow for roughly the same values, which is everything but `null` or `undefined` (given that `strict` mode or `strictNullChecks` is activated).

```
let obj: {}; // Similar to Object  
obj = 32;  
obj = "Hello";  
obj = true;  
obj = () => { console.log("Hello") };  
obj = undefined; // Error  
obj = null; // Error  
obj = { name: "Stefan", age: 40 };  
obj = [];  
obj = /.*/;
```

The `Object` interface is compatible with all values that have the `Object` prototype, which is every value from every *primitive* and *compound* type. However, `Object` is a defined interface in TypeScript, and has some requirements for certain functions. For example, the `toString` method

which is `toString()` => `string` and part of any non-nullish value, is part of the `Object` prototype. If you assign a value with a different `toString` method, TypeScript will error.

```
let okObj: {} = {
  toString() {
    return false;
  }
}; // OK

let obj: Object = {
  toString() {
    return false;
  }
}
// ^-  Type 'boolean' is not assignable to type 'string'.ts(2322)
```

`Object` can cause some confusion due to this behavior, so in most cases, you're good with `{}`.

TypeScript also has a *lowercase* `object` type. This is more the type you're looking for, as it allows for any compound type, but no primitive types.

```
let obj: object;
obj = 32; // Error
obj = "Hello"; // Error
obj = true; // Error
obj = () => { console.log("Hello") };
obj = undefined; // Error
obj = null; // Error
obj = { name: "Stefan", age: 40 };
obj = [];
obj = /.*/;
```

If you want a type that excludes functions, regexes, arrays, and the likes, wait for Chapter 5, where we create one on our own.

2.4 Working with tuple types

Problem

You are using JavaScript arrays to organize your data. The order is important, and so are the types at each position. But TypeScript's type inference makes it really cumbersome to work with it.

Solution

Annotate with tuple types.

Discussion

Next to objects, JavaScript arrays are a popular way to organize data in a complex object. Instead of writing a typical `Person` object as we did in other recipes, you can store entries element by element:

```
const person = ["Stefan", 40]; // name and age
```

The benefit of using arrays over objects is that array elements don't have property names. When you assign each element to variables using destructuring, it gets really easy to assign custom names:

```
// objects.js
// Using objects
const person = {
  name: "Stefan",
  age: 40
}

const { name, age } = person;

console.log(name); // Stefan
console.log(age); // 40

const { anotherName = name, anotherAge = age } = person;

console.log(anotherName); // Stefan
console.log(anotherAge); // 40

// arrays.js
// Using arrays
const person = ["Stefan", 40]; // name and age

const [name, age] = person;
```



```
console.log(name); // Stefan
console.log(age); // 40

const [anotherName, anotherAge] = person;

console.log(anotherName); // Stefan
console.log(anotherAge); // 40
```

For APIs where you need to assign new names constantly, using Arrays is really comfortable, as we see in Chapter 10.

When using TypeScript and relying on type inference, this pattern can cause some issues. By default, TypeScript infers the array type from an assignment like that. Arrays are open-ended collections with the same element in each position.

```
const person = ["Stefan", 40];
// person: (string | number)[]
```

So TypeScript thinks that `person` is an array, where each element can be either a string or a number, and it allows for plenty of elements after the original two. This means when you're destructuring, each element is also of type string or number.

```
const [name, age] = person;
// name: string | number
// age: string | number
```

That makes a comfortable pattern in JavaScript really cumbersome in Typescript. You would need to do control flow checks to narrow down the type to the actual one, where it should be clear from the assignment that this is not necessary.

Whenever you think you need to do extra work in JavaScript just to satisfy TypeScript, there's usually a better way. In that case, you can use tuple types to be more specific on how your array should be interpreted.

Tuple types are a sibling to array types that work on a different semantic. While arrays can be potentially endless in size and each element is of the

same type (no matter how broad), tuple types have a fixed size and each element has a distinct type.

The only thing that you need to do to get tuple types is to explicitly annotate.

```
const person: [string, number] = ["Stefan", 40];

const [name, age] = person;
// name: string
// age: number
```

Fantastic! Tuple types have a fixed length, this means that the length is also encoded in the type. So assignments that go out of bounds are not possible, and TypeScript will throw an error.

```
person[1] = 41; // OK!
person[2] = false; // Error
//^- Type 'false' is not assignable to type 'undefined'.(2322)
```

TypeScript also allows you to add labels to tuple types. This is just meta information for editors and compiler feedback but allows you to be clearer about what to expect from each element.

```
type Person = [name: string, age: number];
```

This will help you and your colleagues to easier understand what to expect, just like object types.

Tuple types can also be used to annotate function arguments. This function

```
function hello(name: string, msg: string): void {
  // ...
}
```

Can also be written with tuple types.

```
function hello(...args: [name: string, msg: string]): {
  // ...
}
```

And you can be very flexible in defining it:

```
function h(a: string, b: string, c: string): void {  
    //...  
}  
// equal to  
function h(a: string, b: string, ...r: [string]): void {  
    //...  
}  
// equal to  
function h(a: string, ...r: [string, string]): void {  
    //...  
}  
// equal to  
function h(...r: [string, string, string]): void {  
    //...  
}
```

This is also known as **rest** elements, something that we have in JavaScript and that allows you to define functions with an almost limitless argument list, where the last element, the **rest** element sucks all excess arguments in.

When you need to collect arguments in your code, you can use a tuple before you apply them to your function.

```
const person: [string, number] = ["Stefan", 40];  
  
function hello(...args: [name: string, msg: string]): {  
    // ...  
}  
  
hello(...person);
```

Tuple types are really useful for many scenarios. We will see a lot more in Chapter 7 and Chapter 10.

2.5 Understanding interfaces vs type aliases

Problem

There are two different ways in TypeScript to declare object types: Interfaces and type aliases. Which one should you use?

Solution

Use type aliases for types within your project's boundary, and use interfaces for contracts that are meant to be consumed by others.

Discussion

Both approaches to defining object types have been subject to lots of blog articles over the years. And all of them became outdated as time progressed. Right now, there is little difference between type aliases and interfaces. And everything that was different has been gradually aligned.

Syntactically, the difference between interfaces and type aliases is nuanced.

```
type PersonAsType = {  
  name: string;  
  age: number;  
  address: string[];  
  greet(): string;  
};  
  
interface PersonAsInterface {  
  name: string;  
  age: number;  
  address: string[];  
  greet(): string;  
}
```

You can use interfaces and type aliases for the same things, in the same scenarios:

- In an implements declaration for classes
- As a type annotation for object literals
- For recursive type structures

There is however one important difference that can have side effects you usually don't want to deal with: Interfaces allow for declaration merging,

but type aliases don't. Declaration merging allows for adding properties to an interface even after it has been declared.

```
interface Person {  
    name: string;  
}  
  
interface Person {  
    age: number;  
}  
  
// Person is now { name: string; age: number; }
```

TypeScript itself uses this technique a lot in `lib.d.ts` files, making it possible to just add deltas of new JavaScript APIs based on ECMAScript versions. This is a great feature if you want to extend e.g. `Window`, but it can fire back in other scenarios. Take this as an example:

```
// Some data we collect in a web form  
interface FormData {  
    name: string;  
    age: number;  
    address: string[];  
}  
  
// A function that sends this data to a back-end  
function send(data: FormData) {  
    console.log(data.entries()) // this compiles!  
    // but crashes horrendously in runtime  
}
```

So, where does the `entries()` method come from? It's a DOM API! `FormData` is one of the interfaces provided by browser APIs, and there are a lot of them. They are globally available, and nothing keeps you from extending those interfaces. And you get no notification if you do.

You can of course argue about proper naming, but the problem persists for all interfaces that you make available globally, maybe from some dependency where you don't even know they add an interface like that to the global space.

Changing this interface to a type alias immediately makes you aware of this problem:

```
type FormData = {  
  // ^-- Duplicate identifier 'FormData'.(2300)  
  name: string;  
  age: number;  
  address: string[];  
}
```

Declaration merging is a fantastic feature if you are creating a library that is consumed by other parts in your project, maybe even other projects entirely written by other teams. It allows you to define an interface that describes your application but allows your users to adapt it to reality if needed. Think of a plug-in system, where loading new modules enhances functionality. Here, declaration merging is a feature you don't want to miss.

Within your module's boundaries, however, using type aliases prevents you from accidentally re-using or extending already declared types. Use type aliases when you don't expect others to consume them.

Performance

Using type aliases over interfaces has sparked some discussion in the past, as interfaces have been considered much more performant in their evaluation than type aliases, even resulting in a performance recommendation on the official [TypeScript wiki](#). This recommendation is meant to be taken with a grain of salt, though, and is as everything much more nuanced.

On creation, simple type aliases may perform faster than interfaces because interfaces are never closed and might be merged with other declarations. But interfaces may perform faster in other places because they're known ahead of time to be object types. Ryan Canavaugh from the TypeScript team expects performance differences to be really measurable with an extraordinary amount of interfaces or type aliases to be declared (around 5000 according to [this tweet](#)).

If your TypeScript code base doesn't perform well, it's not because you declared too many type aliases instead of interfaces, or vice versa

2.6 Defining function overloads

Problem

Your function's API is very flexible and allows for arguments of various types, where context is important. This is hard to type in just a single function signature.

Solution

Use function overloads.

Discussion

JavaScript is very flexible when it comes to function arguments. You can pass basically any parameters, of any length. As long as the function body treats the input right, you're good. This allows for very ergonomic APIs, but it's also very tough to type.

Think of a conceptual task runner. With a `task` function you define new tasks by name, and either passes a callback or pass a list of other tasks to be executed. Or both: a list of tasks that needs to be executed *before* the callback runs.

```
task("default", ["scripts", "styles"]);

task("scripts", ["lint"], () => {
  // ...
});

task("styles", () => {
  // ...
});
```

If you think “this looks a lot like Gulp 6 years ago”, you’re right. Its flexible API where you couldn’t do much wrong was also one of the reasons Gulp was so popular.

Typing functions like this can be a nightmare. Optional arguments, different types at the same position, this is tough to do even if you use union types¹.

```
type CallbackFn = () => void;

function task(name: string, param2: string[] | CallbackFn,
param3?: CallbackFn): void {
    //...
}
```

This catches all variations from the example above, but it’s also wrong, as it allows for combinations that don’t make any sense.

```
task("what", () => {
    console.log("Two callbacks?")
}, () => {
    console.log("That's not supported, but the types say yes!")
})
```

Thankfully, TypeScript has a way to solve problems like this: Function overloads. Its name hints at similar concepts from other programming languages: Defining the same, but with different behavior. The biggest difference in TypeScript, as opposed to other programming languages, is that function overloads only work on a type system level and have no effect on the actual implementation.

The idea is that you define every possible scenario as its own function signature. The last function signature is the actual implementation.

```
// Types for the type system
function task(name: string, dependencies: string[]): void;
function task(name: string, callback: CallbackFn): void
function task(name: string, dependencies: string[], callback:
CallbackFn): void
// The actual implementation
function task(name: string, param2: string[] | CallbackFn,
param3?: CallbackFn): void {
```



```
    //...  
}
```

There are a couple of things that are important to note:

First, TypeScript only picks up the declarations before the actual implementation as possible types. If the actual implementation signature is also relevant, duplicate it.

Also, the actual implementation function signature can't be anything. TypeScript checks if the overloads can be implemented with the implementation signature.

If you have different return types, it is your responsibility to make sure that inputs and outputs match.

```
function fn(input: number): number  
function fn(input: string): string  
function fn(input: number | string): number | string {  
    if(typeof input === "number") {  
        return "this also works";  
    } else {  
        return 1337;  
    }  
}  
  
const typeSaysNumberButItsAString = fn(12);  
const typeSaysStringButItsANumber = fn("Hello world");
```

The implementation signature usually works with a very broad type, which means you have to do a lot of checks that you would need to do in JavaScript anyways. This is good as it urges you to be extra careful.

If you need overloaded functions as their own type, to use them in annotations and assign multiple implementations, you can always create a type alias:

```
type TaskFn = {  
    (name: string, dependencies: string[]): void;  
    (name: string, callback: CallbackFn): void;  
    (name: string, dependencies: string[], callback: CallbackFn):  
    void;  
}
```

As you can see, you only need the type system overloads, not the actual implementation definition.

2.7 Defining this parameter types

Problem

You are writing callback functions that make assumptions of `this`, but don't know how to define `this` when writing the function standalone.

Solution

Define a `this` parameter type at the beginning of a function signature.

Discussion

If there has been one source of confusion for aspiring JavaScript developers it has to be the ever-changing nature of the `this` object pointer.

Sometimes when writing JavaScript, I want to shout “This is ridiculous!”. But then I never know what this refers to.

Unknown JavaScript developer

Especially if your background is a class-based object-oriented programming language, where `this` always refers to an instance of a class. `this` in JavaScript is entirely different, but not necessarily harder to understand. What's, even more, is that TypeScript can greatly help us get more closure about `this` in usage.

`this` lives within the scope of a function, and that points to an object or value that is bound to that function. In regular objects, `this` is pretty straightforward.

```
const author = {  
  name: "Stefan",  
  // function shorthand  
  hi() {
```

```

        console.log(this.name);
    },
};

author.hi(); // prints 'Stefan'

```

But functions are values in JavaScript, and can be bound to a different context, effectively changing the value of `this`.

```

const author = {
  name: "Stefan",
};

function hi() {
  console.log(this.name);
}

const pet = {
  name: "Finni",
  kind: "Cat",
};

hi.apply(pet); // prints "Finni"
hi.call(author); // prints "Stefan"

const boundHi = hi.bind(author);

boundHi(); // prints "Stefan"

```

It doesn't help that the semantics of `this` change again if you use arrow functions instead of regular functions.

```

class Person {
  constructor(name) {
    this.name = name;
  }

  hi() {
    console.log(this.name);
  }

  hi_timeout() {
    setTimeout(function() {
      console.log(this.name);
    }, 0);
  }
}

```

```

    hi_timeout_arrow() {
      setTimeout(() => {
        console.log(this.name);
      }, 0);
    }
  }

  const person = new Person("Stefan")
  person.hi(); // prints "Stefan"
  person.hi_timeout(); // prints "undefined"
  person.hi_timeout_arrow(); // prints "Stefan"

```

With TypeScript, we can get more information on what `this` is and, more importantly, what it's supposed to be through `this` parameter types.

Take a look at the following example. We access a button element via DOM APIs and bind an event listener to it. Within the callback function, `this` is of type `HTMLButtonElement`, which means you can access properties like `classList`.

```

const button = document.querySelector("button");
button?.addEventListener("click", function() {
  this.classList.toggle("clicked");
});

```

The information on `this` is provided by the `addEventListener` function. If you extract your function in a refactoring step, you retain the functionality, but TypeScript will error, as it loses context to `this`.

```

const button = document.querySelector("button");
button.addEventListener("click", handleToggle);

function handleToggle() {
  this.classList.toggle("clicked");
  // ^- 'this' implicitly has type 'any'
  //     because it does not have a type annotation
}

```

The trick is to tell TypeScript that `this` is supposed to be of a specific type. You can do this by adding a parameter at the very first position in your function signature that is named `this`.

```
const button = document.querySelector("button");
button?.addEventListener("click", handleToggle);

function handleToggle(this: HTMLButtonElement) {
  this.classList.toggle("clicked");
}
```

This argument gets removed once compiled. TypeScript now has all the information it needs to make sure you `this` needs to be of type `HTMLButtonElement`, which also means that you get errors once we use `handleToggle` in a different context.

```
handleToggle();
// ^- The 'this' context of type 'void' is not
//     assignable to method's 'this' of type 'HTMLButtonElement'.
```

You can make `handleToggle` even more useful if you define `this` to be `HTMLElement` a super-type of `HTMLButtonElement`.

```
const button = document.querySelector("button");
button?.addEventListener("click", handleToggle);

const input = document.querySelector("input");
input?.addEventListener("click", handleToggle);

function handleToggle(this: HTMLElement) {
  this.classList.toggle("clicked");
}
```

When working with `this` parameter types, you might want to make use of two helper types that can either extract or remove `this` parameters from your function type.

```
function handleToggle(this: HTMLElement) {
  this.classList.toggle("clicked");
}

type ToggleFn = typeof handleToggle;
// (this: HTMLElement) => void

type WithoutThis = OmitThisParameter<ToggleFn>
// () => void
```

```
type ToggleFnThis = ThisParameterType<ToggleFn>  
// HTMLElement
```

There are more helper types when it comes to `this` in classes and objects. See more in [XREF HERE](#) and [XREF HERE](#).

2.8 Working with Symbols

Problem

You see the type `symbol` popping up in some error messages, but you don't know what they mean or how you can use them.

Solution

Create symbols for object properties you want to be unique, and not iterable. They're great for storing and accessing sensitive information.

Discussion

`symbol` is a primitive data type in JavaScript and TypeScript, which, amongst other things, can be used for object properties. Compared to `number` and `string`, `symbol`'s have some unique features that make them stand out.

Symbols can be created using the `Symbol()` factory function:

```
const TITLE = Symbol('title')
```

`Symbol` has no constructor function. The parameter is an optional description. By calling the factory function, `TITLE` is assigned the unique value of this freshly created symbol. This symbol is now unique, distinguishable from all other symbols, and doesn't clash with any other symbols that have the same description.

```
const ACADEMIC_TITLE = Symbol('title')
const ARTICLE_TITLE = Symbol('title')

if(ACADEMIC_TITLE === ARTICLE_TITLE) {
  // This is never true
}
```

The description helps you to get info on the Symbol during development time:

```
console.log(ACADEMIC_TITLE.description) // title
console.log(ACADEMIC_TITLE.toString()) // Symbol(title)
```

Symbols are great if you want to have comparable values that are exclusive and unique. For runtime switches or mode comparisons:

```
// A really bad logging framework
const LEVEL_INFO = Symbol('INFO')
const LEVEL_DEBUG = Symbol('DEBUG')
const LEVEL_WARN = Symbol('WARN')
const LEVEL_ERROR = Symbol('ERROR')

function log(msg, level) {
  switch(level) {
    case LEVEL_WARN:
      console.warn(msg); break
    case LEVEL_ERROR:
      console.error(msg); break;
    case LEVEL_DEBUG:
      console.log(msg);
      debugger; break;
    case LEVEL_INFO:
      console.log(msg);
  }
}
```

Symbols also work as property keys but are not iterable, which is great for serialization

```
const print = Symbol('print')

const user = {
  name: 'Stefan',
  age: 40,
  [print]: function() {
```

```

    console.log(`${this.name} is ${this.age} years old`)
  }
}

JSON.stringify(user) // { name: 'Stefan', age: 40 }
user[print]() // Stefan is 40 years old

```

There's a global symbols registry that allows you to access tokens across your whole application.

```

Symbol.for('print') // creates a global symbol

const user = {
  name: 'Stefan',
  age: 37,
  // uses the global symbol
  [Symbol.for('print')]: function() {
    console.log(`${this.name} is ${this.age} years old`)
  }
}

```

The first call to `Symbol.for` creates a symbol, the second call uses the same symbol. If you store the symbol value in a variable and want to know the key, you can use `Symbol.keyFor()`

```

const usedSymbolKeys = []

function extendObject(obj, symbol, value) {
  //Oh, what symbol is this?
  const key = Symbol.keyFor(symbol)
  //Alright, let's better store this
  if(!usedSymbolKeys.includes(key)) {
    usedSymbolKeys.push(key)
  }
  obj[symbol] = value
}

// now it's time to retrieve them all
function printAllValues(obj) {
  usedSymbolKeys.forEach(key => {
    console.log(obj[Symbol.for(key)])
  })
}

```

Nifty!

TypeScript has full support for symbols, and they are prime citizens in the type system. `symbol` itself is a data type annotation for all possible symbols. See the `extendObject` function from earlier on. To allow for all symbols to extend our object, we can use the `symbol` type:

```
const sym = Symbol('foo')

function extendObject(obj: any, sym: symbol, value: any) {
  obj[sym] = value
}

extendObject({}, sym, 42) // Works with all symbols
```

There's also the sub-type `unique symbol`. A `unique symbol` is closely tied to the declaration, only allowed in `const` declarations and references this exact symbol, and nothing else.

You can think of a nominal type in TypeScript for a very nominal value in JavaScript.

To get to the type of `unique symbol`'s, you need to use the `typeof` operator.

```
const PROD: unique symbol = Symbol('Production mode')
const DEV: unique symbol = Symbol('Development mode')

function showWarning(msg: string, mode: typeof DEV | typeof PROD)
{
  // ...
}
```

At the time of writing, the only possible nominal type is TypeScript's structural type system.

Symbols stand at the intersection between nominal and opaque types in TypeScript and JavaScript. And are the closest things we get to nominal type checks at runtime.

2.9 Understanding value and type namespaces

Problem

You find it confusing that you can use certain names as type annotations, and not others.

Solution

Learn about type and value namespaces, and which names contribute to what.

Discussion

TypeScript is a superset of JavaScript, which means it adds more things to an already existing and defined language. Over time you learn to spot which parts are JavaScript, and which parts are TypeScript.

It really helps to see TypeScript as this additional layer of types upon regular JavaScript. A thin layer of meta-information, which will be peeled off before your JavaScript code runs in one of the available runtimes. Some people even speak about TypeScript code “erasing to JavaScript” once compiled.

TypeScript being this layer on top of JavaScript also means that different syntax contributes to different layers. While a **function** or **const** creates a name in the JavaScript part, a **type** declaration or an **interface** contributes a name in the TypeScript layer.

```
// Collection is in TypeScript land! --> type  
type Collection = Person[]  
  
// printCollection is in JavaScript land! --> value  
function printCollection(coll: Collection) {  
    console.log(...coll.entries)  
}
```

We also say that declarations contribute either a name to the **type** namespace or to the **value** namespace. Since the type layer is on top of the value layer, it's possible to consume values in the type layer, but not vice versa. We also have explicit keywords for that.

```

// a value
const person = {
  name: "Stefan"
}

// a type
type Person = typeof person;

```

`typeof` creates a name available in the type layer from the value layer below.

It gets irritating when there are declaration types that create both types and values. Classes for instance can be used in the TypeScript layer as a type, as well as in JavaScript as a value.

```

// declaration
class Person {
  name: string

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

// used as a value
const person = new Person("Stefan")

// used as a type
type Collection = Person[]

function printPersons(coll: Collection) {
  //...
}

```

And naming conventions trick you. Usually, we define classes, types, interfaces, enums, etc. with a capital first letter. And even if they may contribute values, they for sure contribute types. Well, until you write uppercase functions for your React app, as the convention dictates.

If you're used to using names as types and values, you're going to scratch your head if you suddenly get a good old **TS2749: *YourType* refers to a value, but is being used as a type** error.

```

type PersonProps = {
  name: string
}

function Person({ name }: PersonProps) {
  // ...
}

type PrintComponentProps = {
  collection: Person[],
  //      ^- 'Person' refers to a value,
  //      but is being used as a type
}

```

This is where TypeScript can get really confusing. What is a type, what is a value, why do we need to separate this, and why doesn't this work like in other programming languages? Suddenly, you see yourself confronted with `typeof` calls or even the `InstanceType` helper type, because you realize that classes actually contribute two types (see Chapter 11).

Classes contribute a name to the type namespace, and since TypeScript is a structural type system, they allow values that have the same shape as an instance of a certain class. So this is allowed.

```

class Person {
  name: string

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

function printPerson(person: Person) {
  console.log(person.name);
}

printPerson(new Person("Stefan")); // ok
printPerson({ name: "Stefan" }); // also ok

```

However, `instanceof` checks, which are working entirely in the value namespace and just have implications on the type namespace, would fail, as objects with the same shape may have the same properties, but are not an actual *instance* of a class.

```
function checkPerson(person: Person) {
  return instanceof Person
}

checkPerson(new Person("Stefan")); // true
checkPerson({ name: "Stefan" }) // false
```

So it's good to understand what contributes types, and what contributes value. This table, adapted from the TypeScript docs, sums it up nicely:

Table 2-1. Type and value namespaces

Declaration type	Type	Value
Class	X	X
Enum	X	X
Interface	X	
Type Alias	X	
Function		X
Variable		X

If you stick with functions, interfaces (or type aliases, see [Recipe 2.5](#)), and variables at the beginning, you will get a feeling of what you can use where. If you work with classes, think about the implications a bit longer.

¹ Union types are a way to combine two different types into one. We learn more about union types in Chapter 3

About the Author

Stefan Baumgartner is a developer and architect based in Austria. He is the author of “TypeScript in 50 Lessons” and runs a popular **TypeScript and technology blog**. In his spare-time, he organizes several meetup and conferences, like the Rust Linz meetup and the **European TypeScript conference**. Stefan enjoys Italian food, Belgian beer and British vinyl records.